TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie

Studijní obor: 1802R007 – Informační technologie

Online systém pro zpracování dat o využívání elektrické energie

Online system for evaluation of energy utilisation data

Bakalářská práce

Autor: **Ondřej Smola**

Vedoucí práce: Ing. Jan Kraus, Ph.D.

Konzultant: Ing. Pavel Štěpán

V Liberci dne 17.5.2012

Zadání

SEM VLOŽIT ORIGINÁLNÍ ZADÁNÍ

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Na tomto místě bych rád poděkoval Ing. Janu Krausovi, Ph.D., za odborné vedení bakalářské práce a propůjčení materiálů. Dále bych chtěl poděkovat firmě KMB systems za poskytnutí reálného vzorku dat z elektroměrů. Na závěr bych chtěl poděkovat Ing. Pavlovi Štěpánovi za poskytnuté konzultace.

Abstrakt

Cílem této práce je návrh a implementace webového portálu pro vyhodnocování a vizualizaci dat z elektroměrů s ohledem na možnou budoucí rozšiřitelnost a modularitu. Vývoj byl proveden na základě spolupráce s firmou KMB systems, která během vývoje poskytla dokumentaci a testovací vzorek záznamů z elektroměrů. Samotný vývoj začíná analýzou, zhodnocením aktuálních řešení a požadavků budoucích uživatelů. Na základě získaných informací jsou zvoleny základní technologie pro vývoj. Mezi základní technologii patří platforma Java EE a aplikační rámce Spring, Struts2 a Hibernate. Pro webovou prezentaci je zvolen jazyk JavaScript a technologie Java Server Pages. Zvolené technologie jsou nejprve popsány a je určen způsob jejich využití v aplikaci. Poté je na jejich základě proveden samotný návrh systému. Aplikace je rozdělena na tři moduly, rozdělující samotnou funkcionalitu aplikace na nezávislé části. Je vytvořeno a popsáno schéma vzájemné komunikace mezi moduly a pevně definovány jednotlivé domény problémů, které moduly řeší. Vlastní popis implementace se řídí rozdělením na jednotlivé moduly a jsou zde popsány jejich základní stavební prvky a konfigurace. V implementaci jsou popsány vybrané detaily, které mají zásadní vliv na logiku aplikace. Samotný popis implementace je zakončen konfigurací umístění webové aplikace do webového kontejneru aplikačního serveru a popisem integrace jednotlivých aplikačních rámců. Implementovaná aplikace je poté zprovozněna na zvoleném serveru s poskytnutým vzorkem dat a vyhodnocena rychlost aplikace. V poslední části je poté zhodnocena funkcionalita aplikace a problematika vyplývající z realizovaného návrhu s možnostmi jejího budoucího rozšíření.

Klíčová slova : Vyhodnocování využití elektrické energie, Java EE, Spring, Struts2, Hibernate

Abstract

The aim of this thesis is to design and implement a web portal for visualization and evaluation of data retrieved from electric meters with regard to possible future scalability and modularity. The development was made on the basis of cooperation with KMB systems, which provided documentation and electric meters testing record samples for the development. The actual description of the development begins with analysis and evaluation of current solutions and future requirements from users. Basic technologies for development are selected with regard to gathered information. The basic technologies are the Java EE platform and Spring, Struts2 and Hibernate application frameworks. For Web presentation JavaScript and Java Server Pages is selected. The chosen technologies are firstly described and their method of use in application is defined. The system design is then realized on their actual basis. The application is divided into three modules, that split functionality of the application into independent parts. The scheme of mutual communication between modules is created and described. The individual domains, that modules solves, are defined. Description of implementation accords to the division of the modules and describes their basic building blocks and their configuration. Implementation details that have a major impact on the application logic are described in the implementation part. The actual description of implementation is completed by a description of the location of the web application configuration in the web application server container, and a description of integration for each application framework. The implemented application is then put into operation on the selected server with selected sample data and application performance is evaluated. In the last section the application functionality and issues comming from the design and realized possibilities for future extension of the application are evaulated.

Keywords : Evaluation of the use of electric energy , Java EE, Spring, Struts2, Hibernate

Obsah

Seznam ilustrací

Seznam kódů

Seznam značek, zkratek a termínů

AJAX	Asynchronous JavaScript
API	Application programming interface
CSS	Cascading Style Sheets (Káskádové Styly)
DOM	Document Object Model
DRW	Direct Web Remoting
DTO	Data Transfer Object
GWT	Google Web Toolkit
HQL	Hibernate Query Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JAR	Java Archive
Java EE	Java Enterprise Edition
JDBC	Java Database Connectivity
JPA	Java Persistence API
JS	JavaScript
JSON	JavaScript Object Notation
JSP	Java Server Pages
MVC	Model-view-controller
OGNL	Object-Graph Navigation Language
ORM	Object-relational mapping (Objektově-relační mapování)
PDF	Portable Document Format
POJO	Plain Old Java Object
RIA	Rich Internet Application
SQL	Structured Query Language
URL	Uniform Resource Locator
WAR	Web Archive
XML	Extensible Markup Language

Úvod

S rozvojem informačních technologií ve firmách vzniká velká poptávka po softwaru, který usnadní a zpřístupní produkty koncovým zákazníkům. Mezi rozšířené typy programů patří systémy, které umožní složitá data prezentovat ve formě srozumitelné pro veřejnost. S rozvojem internetu a chytrých mobilních telefonů dále rostou požadavky na přístupnost online a klesá poptávka po klasických offline systémech, které uživatele nutí instalovat aplikace do vlastního počítače a obvykle vyžadují i větší technickou zdatnost uživatele.

Velmi oblíbené jsou portály umožňující sledovat stav zařízení online, od meteorologické stanice až po komplexní systémy řízení telefonních ústředen. Obrovská výhoda tohoto řešení tkví v přístupnosti a jednoduchosti. K zobrazení dat již nepotřebujeme výkonný hardware ani specifický software na straně klienta. Oproti tomu dochází k velkému nárůstu požadavků na straně webové aplikace, jak na použité technologie a celkovou rychlost, tak na celkovou kvalitu návrhu, který musí být schopen reagovat na vznikající podněty pro rozšíření a úpravy.

Cílem práce je vytvořit webový portál umožňující sledovat a vyhodnocovat data z elektroměrů firmy KMB systems pro stávající a budoucí zákazníky. Firma KMB systems se zabývá vývojem elektroniky zejména pro průmyslovou automatizaci a energetiku. Vyhodnocením dat získaných z elektroměrů lze sledovat dlouhodobé trendy ve spotřebě elektrické energie a na jejich základě lze zjistit místa vyžadující optimalizaci rozložení spotřeby. Samotná optimalizace spočívá jak v přesunu činnosti zařízení do levnějších tarifů, tak i přeskupením sledovaných zařízení do jiných skupin. Porovnáváním trendů je navíc možné pružně reagovat na vzniklé změny ve spotřebě a vytvářet dlouhodobé plány pro regulaci spotřeby elektrické energie. Elektroměry mohou zároveň sloužit k sledování množství elektrické energie dodané do sítě. V tomto případě je důležité mít možnost zobrazit nejen množství, ale také vyhodnotit poměr energie dodané vůči získané a zjistit množství ušetřených finančních prostředků.

Mezi hlavní výhody svázanosti aplikace s technologiemi vybrané firmy patří možnost okamžitého zprovoznění pro zákazníka, jelikož není třeba systém konfigurovat na přesný typ dat ze zvolených elektroměrů. Aby bylo možné aplikaci libovolně upravovat, je nutné vytvořit vysoce modulární systém založený na aktuálních webových

technologiích, s co nejmenšími závislostmi na použitém hardware a software. Systém musí být připraven i na poměrně rozsáhlé změny v návrhu, jakými jsou například přechod na distribuované transakce nebo možnost načítat části systému za běhu aplikace.

V první kapitole je popsána analýza požadavků firmy KMB systems, jejich stávající systém a jeho možnosti. Dále je zhodnocena analýza konkurenčních řešení. Ze získaných informací je proveden výběr částí, které je vhodné implementovat v první fázi vývoje, a určité zásady, kterými se bude vývoj řídit.

Ve druhé kapitole je vysvětlen návrh a výběr technologií použitých během vývoje. Jsou zde popsány základní stavební části jednotlivých rámců,je probráno rozdělení projektu na moduly a návrh jejich vzájemné komunikace. Následující tři kapitoly se zabývají popisem jednotlivých modulů a implementací jejich navržené funkcionality. V poslední kapitole je zhodnoceno implementované řešení spolu s nastíněním možností jeho budoucího rozšíření.

1 Analýza požadavků a možnosti jejich řešení

Aplikace vznikala na základě podkladů a konzultací poskytnutých firmou KMB systems [1]. Na jejich základě byl vytvořen model jejího možného budoucího komerčního využití, a proto byla aplikace od počátku vyvíjena a testována na vzorcích reálných dat. Návrh aplikace probíhal na základě požadavků potencionálních zákazníků a informací od zaměstnanců firmy KMB systems. Značnou výhodou při analýze byly připomínky a nápady, které vznikly již během vývoje původní aplikace ENVIS a poznámky od jednoho z potencionálních zákazníků právě vyvíjené aplikace. Tyto informace velmi pomohly k ucelení základního obrazu o prvotních požadavcích na aplikaci. Mezi hlavní požadavky patří možnost správy dat více společností a firem a s tím související možnost hierarchické uspořádání elektroměrů do skupin. Základními informacemi pro uživatele bude rychlý přehled o stavu jednotlivých přístrojů ve vybraných intervalech a tabulkový výpis jednotlivých hodnot. Uživatel by měl mít možnost porovnat měřené úseky v rámci přístroje a vytvořit zprávu s přehledem za určité období.

1.1 Program ENVIS

Program ENVIS [1] představuje komplexní řešení pro vyhodnocování měření kvality sítě či efektivity využívání elektrické energie. Archivuje a zpřístupňuje uživatelům všechna dostupná data. Poskytuje také jednoduchý nástroj pro konfiguraci, dohled a vzdálenou správu měřicích přístrojů. Data z přístrojů lze archivovat v souborech či ukládat v databázích SQL serveru. Aplikace je psaná pro platformu Microsoft .NET Framework a hlavním programovacím jazykem je jazyk C#. Při vývoji aplikace bylo využito komerční řešení od firmy DXperience [2]. Z velkého množství nabízených knihoven je především využita knihovna XPO, která poskytuje podporu pro objektově relační mapování.

1.2 Dostupná řešení

Mezi aplikace se stejným zaměřením patří portál firmy Northern Design Metering Solutions [3]. Jelikož zmíněný portál umožňuje i demo přístup, bylo možné zhodnotit celkovou funkcionalitu aplikace a získat tak představu o základních bodech návrhu spolu s inspirací pro místa, která by mohla být vylepšena. Portál je již minimálně 2 roky

v provozu a v době psaní této práce byl dostupný ve verzi 2.13. Grafický návrh řešení je znázorněn na obrázcích 16 a 17. V návrhu konkurenčního řešení bylo zjištěno několik nedostatků.

Mezi hlavní nedostatky stávající verze patří neprovedené optimalizace a nepřehledný návrh kódu jazyka JavaScript. Soudě dle grafického návrhu také nepřehlednost v případě více zařízení a neošetřené stavy, kdy nejsou k dispozici žádná zařízení. Mezi hlavní klady aplikace patří kompaktní a barevně příjemný návrh poskytující rozsáhlou funkcionalitu.

Danou problematikou se dále zabývají například firmy portály sMeasure [4] a Opower [5]. Zaměření posledních dvou řešení je spíše pro menší skupiny zařízení nebo jednotlivé objekty. Z těchto důvodů bylo rozhodnuto vytvořit návrh schopný pružně reagovat jak na požadavky pro zjednodušení v případě menších skupin sledovaných zařízení, tak na komplexní uskupení více budov s desítkami zařízení.

1.3 Základní plán vývoje

Na základě získaných poznatků byly sestaveny základní body funkcionality tak, aby bylo možné je navrhnout a implementovat v stanoveném časovém období. Aplikace by měla v tomto stavu sloužit jako prezentace možností a návrhu uživatelského rozhraní, s tím že bude dále rozšiřována.

Základní stanovené cíle lze shrnout do následujících bodů:

1. Stromový výpis zařízení
2. Základní přehled o vybraném zařízení
3. Možnost porovnání vybraných intervalů v rámci zařízení
4. Tabulkový přehled jednotlivých odečtů
5. Vytvoření reportu pro vybrané zařízení
6. Správa více uživatelů

Ze základní implementace byla vynechána možnost porovnání a zobrazení skupin, jelikož tato funkcionalita nebyla implementována v dodané testovací verzi databáze a dle vytvořeného návrhu by se její implementace nevešla do časového plánu.

2 Návrh aplikace

2.1 Java EE

Java EE [6] je moderní a rychle se rozvíjející platforma a spolu s technologií .NET [7]od firmy Microsoft tvoří současnou špičku pro vývoj informačních a podnikových systémů. Platforma Java EE je tvořena velkým množství technologií. Na jejich základě je vytvořeno značné množství placených i volně dostupných rámců, usnadňujících vývoj pro jednotlivé domény. Mezi velmi oblíbené rámce patří například Hibernate, Spring, Struts2, Seam, nebo GWT [8]. Aplikace Java EE využívá pro svůj běh webový kontejner aplikačního serveru. Pro vývoj nativní Java EE aplikace je třeba použít server s plnou specifikací pro Java EE. Mezi rozšířené aplikační servery [8] patří WebLogic, ORACLE GlassFish, JBoss nebo IBM Websphere. Jejich nevýhodou je obvykle delší doba startu, složitější rozhraní a drahé komerční licence. Jelikož tato aplikace využívá rámce Spring a Struts2, je možné využít odlehčený webový kontejner, kdy plně postačuje pouze podpora pro Java Servlet API verze 3 [9]. Mezi nejrozšířenější zástupce odlehčených webových kontejnerů patří Apache Tomcat [10], který byl také použit během vývoje ve verzi 7.

2.2 Rámec Spring

Rámec Spring [11] vznikl roku 2003 jako reakce na problematiku při vývoji Java EE aplikací. Jeho hlavním cílem je nezávislost jednotlivých komponent aplikace a co nejmenší provázání aplikační logiky. Během vývoje aplikace byla nejprve použita verze 3.0, která byla později nahrazena za aktuální verzi 3.1. Při změně verze byla poupravena část logiky pro využití nových možností .

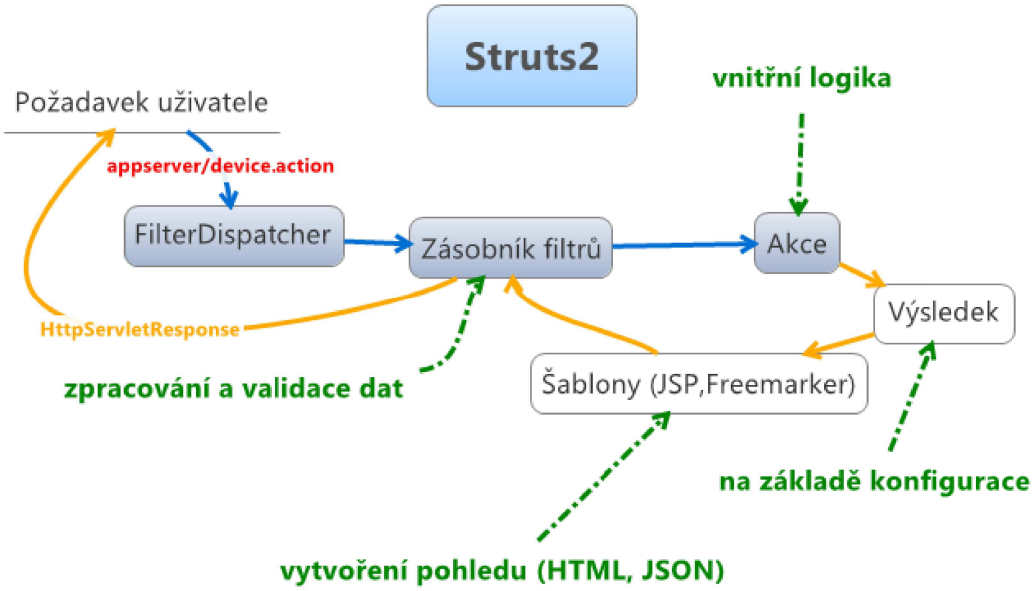
Mezi klíčové vlastnosti rámce Spring patří jeho rozšiřitelnost a modularita. Celý rámec je založen na velkém množství komponent, které lze přidávat a konfigurovat během vývoje. Mezi základní, z více než desítky dostupných, patří Spring Core, Spring Bean a Spring Context. Rámec Spring využívá pro realizaci logiky obyčejné Java třídy, které nejsou nijak svázány implementací určeného rozhraní nebo abstraktní třídy. Instance jednotlivých závislostí třídy jsou poskytnuty až za běhu z kontextu rámce Spring. Pro získání objektů, s kterými instance spravovaná rámcem Spring pracuje, je použit návrhový vzor Inversion of Control, v rámci Spring je často označován jako Dependency

Injection. Inversion of Control je způsob sdílení a přidělování instancí třídy na základě sdíleného kontextu rámce Spring. Rámec Spring umožňuje dvojí způsob konfigurace, první způsob využívá XML konfiguraci a druhý Java anotace.

2.3 Rámec Struts2

Jedná se o moderní odlehčený rámec postavený na návrhovém vzoru MVC. Princip rámce Struts2 [12] vychází, podobně jako u rámce Spring, ze snahy minimalizovat závislosti mezi jednotlivými komponentami aplikace. Rámec umožňuje jednoduchou rozšiřitelnost pomocí pluginů. Již v základním nastavení poskytuje velmi kvalitní podporu pro tvorbu RIA pomocí technologie AJAX. Třídy kontroléru se nazývají akce. Reprezentace modelu je řešena abstraktně a lze použít libovolné stávající řešení, které je v tomto případě reprezentováno rámcem Spring.

2.3.1 Životní cyklus dotazu na server rámce Struts2



Obrázek 1: Schéma dotazu na server realizovaného rámcem Struts2

Hlavním prvkem rámce Struts2 je *FilterDispatcher* [12], jedná se o filtr obalující logiku celého volání akce. Pří příchodu uživatelského požadavku nejprve kontaktuje *ActionMapper*, který na základě dotazované adresy vyhledá příslušnou akci. Kolem akce je vytvořena proxy a kontext. Na základě kontextu proběhne volání definovaného

zásobníku filtrů, podle návrhového vzoru příkaz, pro všechny jeho filtry a cílovou akci. Na základě nastavení je poté z výsledného řetězce vytvořen výsledek ze stránek JSP. Nakonec jsou zavolány jednotlivé filtry v obráceném pořadí a výsledek odeslán zpět uživateli. FilterDispatcher (obrázek 1) ukončí svou činnost odstraněním všech vyvolaných vláken a vyčištěním aktivního kontextu.

2.3.2 Filtr

Jedná se o implementaci návrhového vzoru Intercepting Filter. Filtry jsou rozděleny do pojmenovaných zásobníků, které jsou poté aplikovány na jednotlivé akce. Veškerá nastavení filtrů jsou umístěna v konfiguračních XML souborech rámce. Mezi základní úkoly filtrů patří úprava vstupu a výstupu jednotlivých akcí během zpracování požadavků. Filtry mohou v případě chyby zabránit spuštění samotné akce. Mezi základní filtry patří například *Servlet Config*, který realizuje nastavení části kontextu akce na základě implementovaného rozhraní.

2.3.3 Akce

Základní jednotkou rámce Struts2 je akce. Akce je třída s alespoň jednou metodou, která nemá žádné parametry a návratovou hodnotou je řetězec. Dle konvencí se pro název metody používá *execute*, ale název lze změnit nebo vyhodnotit za běhu pomocí předdefinovaného výrazu. Třídy jednotlivých akcí obvykle dědí od abstraktní třídy *ActionSupport*, která implementuje základní rozhraní a konstanty používané pro vnitřní logiku a návratové hodnoty akce.

2.4 Rozdělení projektu na moduly

Projekt je rozdělen na 3 moduly s názvy Appserver, Core a Core_External, provázaných na základě rozhraní, sdílených knihoven a nastavení. Během vývoje bylo využito mnoha knihoven, které mají následovně množství závislostí na další knihovny, a bylo by velmi složité řešit je ručně. Proto byla velmi důležitá volba kvalitního integračního nástroje. Na základě předchozích zkušeností byl zvolen program Apache Maven ve verzi 3 [13] Jedná se o velmi rozšířený a populární integrační nástroj, který obsahuje spoustu modulů pro automatizaci většiny problémů spojených se sestavením

aplikace a správou aplikace. Základní konfigurační jednotkou systému Maven je soubor *pom.xml*. Samotný popis nastavení je mimo rozsah této práce a je dostupný na stránkách aplikace [13]. Celý soubor projektů byl definován pomocí hlavního konfiguračního souboru *pom.xml*, obsahujícího informace pro jednotlivé moduly, sdílené knihovny, použité pluginy a sdílená úložiště. Hlavní soubor dále obsahuje nastavení verze aplikace, její pojmenování a určení výsledného archivu pro kompilaci. Jednotlivé moduly dále obsahují vlastní konfigurační soubory s definicemi privátních knihoven a zásuvné moduly aplikované pouze na zvolený modul. Hlavním modulem je Appserver. Při sestavení modulu Appserver je vytvořen WAR a ostatní subprojekty jsou přidány při sestavení jako závislosti ve formátu JAR.

2.4.1 Modul Appserver

Modul Appserver obsahuje komponenty pro realizaci webové prezentace, dále pak nastavení a logiku komunikace s interní částí aplikace přes rozhraní modulu Core.

Mezi základní komponenty patří :

1. Popis pro umístění na server (soubor *web.xml*)
2. Nastavení aplikačního kontextu rámce Spring
3. Logika a nastavení rámce Struts2
4. JSP stránky, kaskádové styly, soubory jazyka JavaScript
5. Soubory pro internacionalizaci a nastavení vlastností některých knihoven

2.4.2 Modul Core

Modul Core obsahuje logiku pro zpracování a obsluhu dotazů modulu Appserver. V případě dotazu na externí databázi kontaktuje rozhraní modulu Core_External pro potřebná data.

Mezi základní komponenty patří:

1. Logická vrstva aplikace (procesory dat)
2. Přístup do interní databáze pomocí rámce Hibernate
3. Poskytování služeb pro práci s daty interní databáze
4. Podpůrná logika (vytváření reportů)

2.4.3 Modul Core_External

Modul Core_External slouží pro obsluhu požadavků na data v externí databázi a zajišťuje jejich převod z relační do objektové reprezentace v jazyce Java.

Mezi základní komponenty patří :

1. Modelové třídy pro reprezentaci dat
2. Logika pro abstrakci práce s databází
3. Třídy pro mapování relačních dat do jejich objektové reprezentace

2.5 Použité databáze

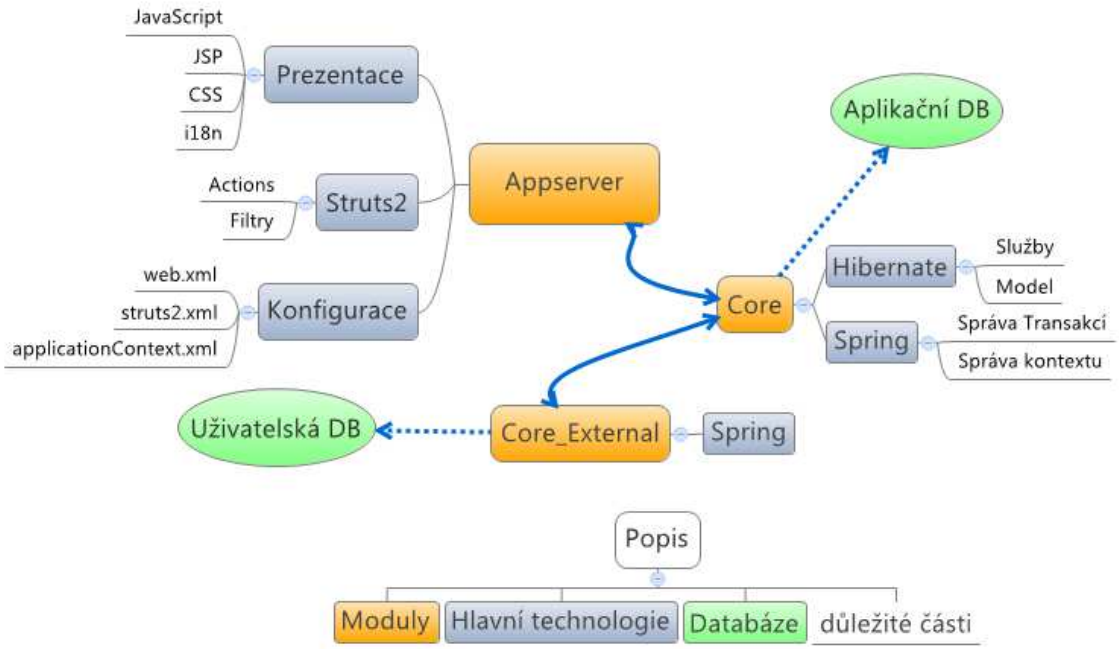
Aplikace bude pracovat s dvěma databázemi. První databáze je nazývána interní, případně aplikační. Jedná se o databázi výhradně spravovanou samotnou aplikací, do které není možné během spuštěné aplikace jinak zasahovat. Pro přístup do interní databáze je použit přístup pomocí rámce Hibernate [14]. V této aplikaci je interní databáze reprezentována pomocí MSSQL databázového serveru. Druhá databáze je nazývána externí nebo uživatelská. K této databázi je zpravidla přistupováno pomocí nízké úrovně abstrakce pomocí jazyka SQL. Během vývoje byl použit pro reprezentaci externí databáze také MSSQL server z důvodu přímé kompatibility s programem Envis. Během vývoje byly tyto databáze odděleny pomocí rozdílného názvu databázové schématu.

2.6 Komunikace mezi moduly

Nejprve bylo třeba pevně definovat způsob komunikace mezi jednotlivými moduly. Jelikož bylo nutné odstínit jednotlivou funkcionalitu mezi moduly a poskytnout pevné body, přes které bude probíhat komunikace, byl zvolen návrhový vzor fasáda, který efektivně řeší danou situaci.

Nejprve vznikne požadavek od uživatele. Po zpracování a validaci dat pomocí filtrů rámce Struts dojde k autentizaci uživatele a vykoná se vnitřní logika akce. Pokud jsou pro vytvoření odpovědi potřeba data, která nejsou přítomna v kontextu akce, je vytvořen požadavek na rozhraní modulu Core. Modul Core provede ověření požadavku na úrovni aplikační databáze a poté provede volání rozhraní modulu Core_External. Modul Core_External poté na základě volání provede dotaz do uživatelské databáze a výsledek

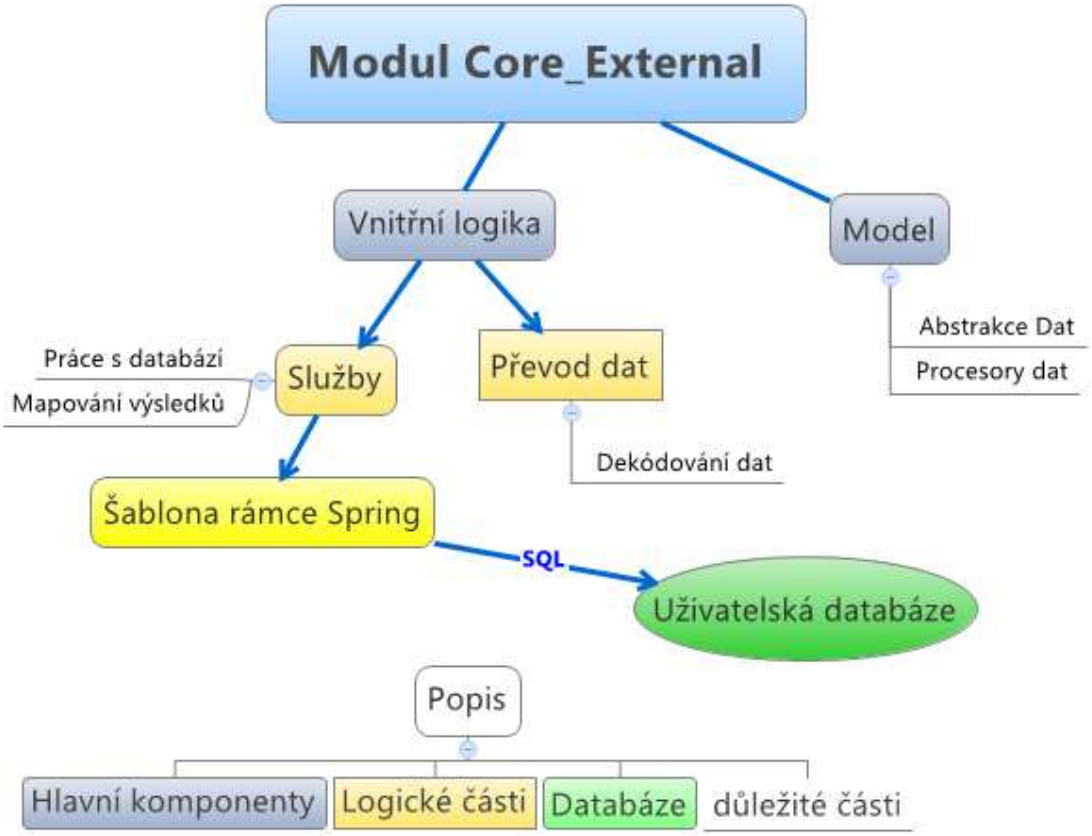
namapuje do jednotlivých instancí Java tříd. Pokud jsou některá data zakódovaná, je zde provedeno také dekódování dat. Data jsou poté předána zpět do modulu Core, který provede abstrakci dat pomocí procesorů dat. Poté je vytvořená instance procesoru vrácena na modul Appserver, který vykoná zbývající logiku a zobrazí data uživateli.



Obrázek 2: Popis jednotlivých modulů včetně hlavních použitých technologií

Princip samotného návrh je zobrazen na obrázku 2.. Efektivita tohoto řešení spočívá v nezávislosti a abstrakci komunikace. Jednotlivé moduly nejsou provázány a jejich vnitřní úpravy nevyžadují změnu logiky v ostatních modulech. Přítomnost kontejneru rámce Spring navíc odstraňuje závislosti i mezi logikou ostatních tříd ve stejném modulu.

3 Modul pro komunikaci s uživatelskou databází



Obrázek 3: Návrh modulu pro zpracování dat z uživatelské databáze

Modul Core_External souží k obsluze požadavků, které pracují s daty uloženými v uživatelské databázi. Jelikož uživatelská databáze může být upravována a nebylo vhodné vytvářet příliš pevné požadavky na její podobu, nebyl zvolen přístup na úrovni objektově relačního mapování (ORM).

3.1 Šablona JDBC rámce Spring

V modulu Core_External se přistupuje k databázi pomocí šablony rámce Spring. Konkrétně byla použita šablona *JdbcTemplate* [15]. Nastavení šablony bylo provedeno deklarativně, na základě zdroje dat s nastavením vlastností připojení, v souboru *persistence.xml*. Samotná instance šablony je bezpečná z hlediska vícevláknového přístupu a proto bylo možné použít jen jednu instanci šablony pro celý modul, sdílenou pomocí

kontextu rámce Spring. Výhoda šablony spočívá v abstrakci samotného volání, jelikož není třeba řešit ošetření výjimek, protože jsou obsluhovány samotnou šablonou. Volání je proto velmi přehledné a srozumitelné. Šablona dále poskytuje pokročilou podporu pro ochranu proti SQL injektáži pomocí pojmenovaných map SQL parametrů. Mapování dat uložených v relační databázi do jejich objektové reprezentace bylo realizováno pomocí tříd pro mapování.

```
<bean id="externalDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${dataSource.envis.external.driver}" />
    <property name="url" value="${dataSource.envis.external.url}"></property>
    <property name="username" value="${dataSource.envis.external.username}" />
    <property name="password" value="${dataSource.envis.external.password}" />
</bean>

<bean name="jdbcTemplate"
    class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg>
        <ref bean="externalDataSource" />
    </constructor-arg>
</bean>
```

Kód 1: Nastavení šablony rámce Spring pomocí XML konfigurace

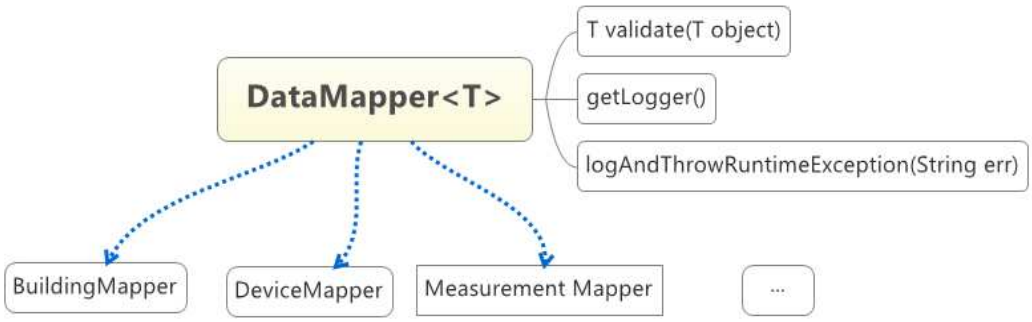
Nastavení a vytvoření šablony bylo provedeno deklarativně pomocí kontextu rámce Spring (kód 1). Při nastavení zdroje dat byl použit výraz, který na základě úložiště vlastností vybere příslušnou hodnotu z externího souboru s nastaveními a pomocí přístupových metod ji přiřadí k nově vytvořené instanci. Soubor s nastavením obsahuje pevně danou strukturu tvaru *název* = *hodnota*. Samotné použití šablony rámce Spring je znázorněno v kódu 2.

```
MapSqlParameterSource parameters = new MapSqlParameterSource();
parameters.addValue("list", configList);

List<DeviceElmerData> result = jdbcTemplate.query(
        "SELECT Config.Id,Install.MTP,Install.MTN FROM SmpConfigsDb Config " +
        "INNER JOIN SmpInstallConfigDB Install " +
        "ON  Config.installConfig = Install.Id WHERE Config.id in (:list)", parameters,
        new DeviceElmerDataMapper());
```

Kód 2: SQL dotaz pomocí šablony rámce Spring používající mapu SQL parametrů

3.2 Třídy pro mapování objektů z relační podoby

Základním stavebním prvkem modulu jsou třídy pro mapování dat získaných z uživatelské databáze. Obvykle jsou tyto objekty označovány jako DTO a typická pro tyto objekty je absence jakékoliv vnitřní logiky. Pokud by byl použit přístup na základě ORM, obsahoval by výsledek dotazu do databáze již vytvořené instance těchto tříd a validace by proběhla na základě konfigurace pomocí nastavení pomocí XML souboru nebo anotací. V případě použití šablon rámce Spring bylo třeba mapování a validaci řešit vlastní logikou. K mapování byly použity instance třídy implementující rozhraní *RowMapper*, obsahující logiku pro vytvoření a validaci nové instance. Jelikož modul mapuje poměrně rozsáhlou databázi, bylo třeba nastavit jednotný způsob implementace tříd využívajících toto rozhraní. Proto byla vytvořena abstraktní třída *DataMapper*, implementující generické rozhraní *RowMapper<T>*, která podporuje definici způsobu validace a umožňuje záznam událostí.



Obrázek 4: Schéma návrhu tříd pro mapování dat z uživatelské databáze

Validace dat je přímo součástí mapování pro zajištění konzistence dat. Hlavním účelem validační logiky je kontrola přítomnosti požadovaných klíčů a validace dat na přípustné hodnoty. Pokud dojde k chybě při validaci, ostatní výsledky se již nezpracovávají a celá operace končí voláním metody *logAndThrowRuntimeException ( )*. Vzniklá výjimka je poté dále zachycena ve filtru zásobníku Struts2 akce a uživateli se zobrazí pouze standardní chybová hláška. Tento způsob řešení poskytuje velké množství informací pro diagnostiku vzniklého problému, jelikož chyba je zaznamenána jak na úrovni modulu Core_External, s přesným chybovým hlášením a stavem zásobníku, tak na úrovni požadavku uživatele během vykonávání akce.

```
public class BuildingMapper extends DataMapper<BuildingReference>{

    @Override
    public BuildingReference mapRow(ResultSet rs, int row) throws SQLException {
        return validate(new BuildingReference(rs.getLong("Id"), rs.getNString("objekt")));
    }

    @Override
    protected BuildingReference validate(BuildingReference dev) {

        if (dev.getId() <= 0)
            logAndThrowRuntimeException("Id cannot be null or <= 0");

        if (StringUtils.isBlank(dev.getName()))
            logAndThrowRuntimeException("device group name cannnot be empty");

        return dev;
    }

}
```

Kód 3: Třída pro mapování a validaci dat do objektové reprezentace

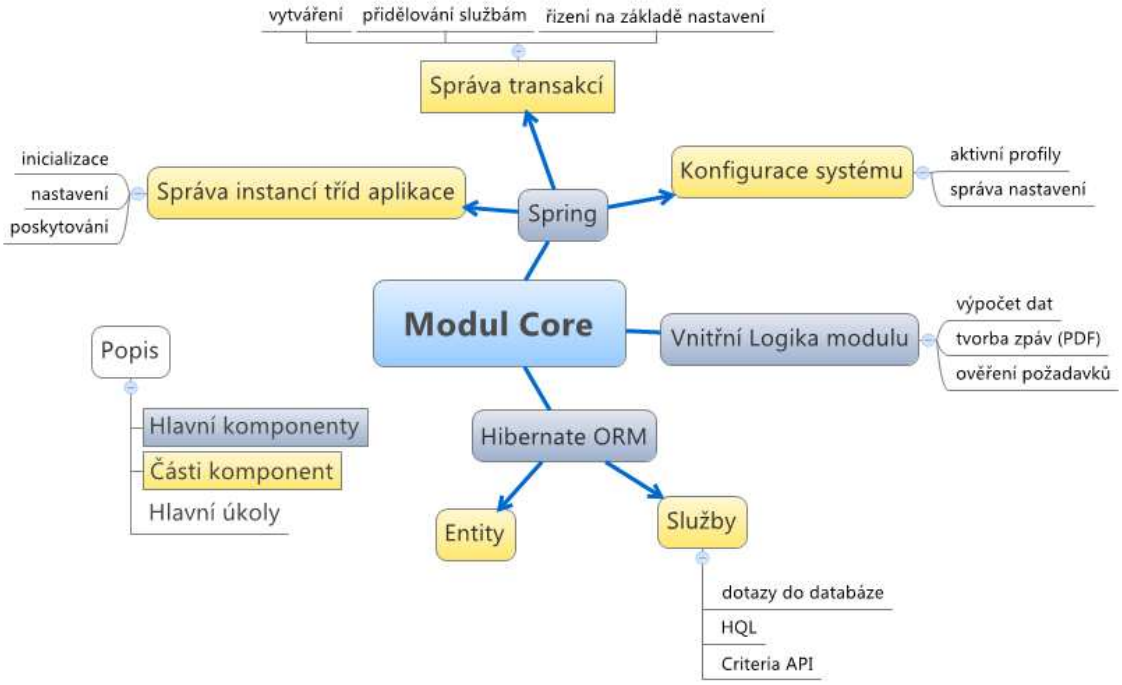
V kódu 3 je znázorněna logika mapování objektů. Pomocí volání metod rozhraní *ResultSet* jsou získány požadované hodnoty vybraných záznamů z databáze. Toto řešení s sebou nese jistá omezení. První omezení spočívá v problematice konverze datových typů, jelikož je předem nutné znát požadovaný datový typ. Další omezení spočívá ve svázanosti s použitou databází, jelikož například pro textový řetězec u MySQL databáze je použita metoda *getString( )* a u MSSQL databáze metodu *getNString( )*. Mezi hlavní výhody patří snadná a jednoduchá realizace a úprava. Po vytvoření instance objektu je reference nejprve předána k validaci a následně je navrácena z volání metody *mapRow( )*. Ověřená instance poté slouží jako součást návratové hodnoty SQL dotazu.

```
@Override
public DeviceProperty findDevice(long deviceId) {

    return jdbcTemplate.queryForObject("select * from SmpIdentifyDB where Id = ?",
            new DeviceMapper(), deviceId);
}
```

Kód 4: SQL dotaz pomocí šablony rámce Spring s ochranou proti SQL injektáži

Kód 4 ilustruje výsledné použití včetně ochrany databáze proti SQL injektáži. U většího počtu parametrů by byla použita mapa SQL parametrů, kde je místo otazníků použit jako klíč název pojmenovaného parametru (kód 2).

4 Vnitřní logika aplikace



Obrázek 5: Návrh vnitřní logiky aplikace s použitím rámců Spring a Hibernate

V modulu Core je umístěna hlavní logika celé aplikace. Modul je rozdělen pomocí Java balíčků na komponenty. Mezi základní komponenty patří datové třídy, služby pro práci s datovými třídami a podpůrná logika pro integraci externích knihoven.

Datová vrstva obsahuje třídy pro přenos dat mezi komponentami a moduly, dále je tvořena entitami pro rámec Hibernate a třídami pro práci s daty ve formátu XML. Mezi základní implementace modelových tříd patří DTO. Jedná se o třídy, které neobsahují žádnou vlastní logiku ani uživatelská metadata. Data jsou obvykle vytvořena v určitém kroku logiky a jelikož byla získána z určitého kontextu, je nutné zaručit jejich neměnnost. Proto jsou jejich členské proměnné obvykle deklarovány jako finální a je možné k nim pouze přistupovat.

```
public class ComparisionResult {

    private final Date actualTime;
    private final Date lastTime;

    private final double actualValue;
    private final double lastValue;

    public ComparisionResult(Date actualTime, Date lastTime, double actualVal, double lastVal) {
        this.actualTime = new Date(actualTime.getTime());
        this.lastTime = new Date(lastTime.getTime());
        this.actualValue = actualVal;
        this.lastValue = lastVal;
    }

    public Date getActualTime() {
        return new Date(actualTime.getTime());
    }
```

Kód 5: Třída pro přenos dat (DTO)

Další části datového modelu jsou perzistentní entity. Jedná se o třídy, které používá rámec Hibernate během interakce s databází. Informace o entitách získává rámec Hibernate na základě reflexe. Rámec Hibernate vytvoří pomocí anotací a datových typů dotaz na vytvoření tabulky v databázi mapující daný objekt do jeho relační podoby. Na základě anotací rámec Hibernate vytvoří validační pravidla, která rozhodují, zda požadovaný záznam splňuje podmínky. V případě získání nevalidního záznamu dotaz končí výjimkou a transakce je dle konfigurace ukončena.

```
@Entity
public class PersonBuildingAssociation extends IdedEntity {

    @ManyToOne(optional = false, targetEntity = Person.class)
    private Person person;

    @ManyToOne(optional = false, targetEntity = Building.class)
    private Building building;

    @NotNull
    @Enumerated(EnumType.STRING)
    private Permission highestPermission;

    @SuppressWarnings("unused")
    private PersonBuildingAssociation() {
    }
```

Kód 6: Entita rámce Hibernate anotovaná pro přístup na základě reflexe

Při vytváření entit je pro Hibernate klíčová anotace *@Entity*. Její přítomnost určuje, zdali se daná třída bude mapovat do databáze. Mapování jednotlivých sloupců tabulky lze provádět pomocí přístupu na základě vlastností, kde se mapování umisťuje nad přístupové

metody. Tento způsob se ale jevil jako méně přehledný, a proto byla zvolena možnost mapovat jednotlivé sloupce přímo u jejich definice pomocí přístupu na základě reflexe.

V kódu 6 je realizace entity vytvářející asociaci mezi entitami *Person* a *Building*. Reference na třídu *Person* a *Building* je mapována jako M:1 a parametr *optional* definuje pravidlo, kdy validní záznam musí obsahovat reference na obě entity. Každá entita dědí od třídy *IdedEntity*. Třída *IdedEntity* definuje základní způsob tvorby primárních klíčů jednotlivých záznamů. V této aplikaci byla logika tvorby identifikátorů a jejich uchovávání plně ponechána na rámci Hibernate, který volí způsob generování unikátního identifikátoru na základě standardního nastavení dle použitého poskytovatele databáze. V případě MySQL a MSSQL serveru je vytvořena unikátní sekvence v databázi, která je společná pro všechny záznamy.

```
@ManyToMany(fetch=FetchType.EAGER,targetEntity=Measurement.class)
@JoinTable(name = "Device_Measurements",
           joinColumns = { @JoinColumn(name = "device_id") },
           inverseJoinColumns = { @JoinColumn(name = "measurement_id") })
@Fetch(FetchMode.SUBSELECT)
private Set<Measurement> measurements = new HashSet<>();
```

Kód 7: Mapování relace M:N pomocí Hibernate

V kódu 7 je znázorněna implementace mapování *M:N* relace pomocí Hibernate. Jedná se o mapování situace, kdy jedno zařízení má více měření, která mohou být zároveň sdílená mezi více zařízeními. Parametr *fetch* určuje strategii pro načítání referenčních objektů. Strategie *FetchType.EAGER* provede načtení referenčních objektů při načtení hlavního objektu podle zvoleného módu *FetchMode*. Standardní mód je *FetchMode.SELECT*, kdy pro každý element kolekce je vytvořen nový dotaz. Zvolený mód *FetchMode.SUBSELECT* nejprve načte hlavní objekt a poté vytvoří jeden nový dotaz pro všechny elementy kolekce. Při tomto nastavení je vhodné určit i maximální možný počet záznamů načítaných pomocí jednoho dotazu. V případě strategie *FetchType.LAZY* je poskytnuta pouze proxy kolekce s identifikátory a jednotlivé reference se načítají až při přístupu k nim.

Poslední důležitou částí datové vrstvy jsou třídy, které slouží pro práci s daty ve formátu XML a JSON. Pro tento účel byla v aplikaci použita knihovna *XStream* [16]. Mezi její hlavní výhody patří možnost použít jako vzor pro serializaci a deserializaci standardní POJO třídy s anotacemi. Dále poskytuje jednoduchou podporu pro serializaci

do formátu JSON, který je mnohem úspornější z hlediska délky výsledného textu a tudíž výhodnější pro přenos pomocí protokolu HTTP. Navíc je data dále možné komprimovat například pomocí GZIP. Nastavení mezi typem zpracovávaného formátu je provedeno během vytváření instance a ostatní kód zůstává nezměněn.

```
XStream xmlStream= new XStream();
xmlStream.processAnnotations(DeviceHierarchy.class);

XStream jsonStream = new XStream(new JsonHierarchicalStreamDriver());
jsonStream.processAnnotations(DeviceHierarchy.class);
jsonStream.setMode(XStream.NO_REFERENCES);
```

Kód 8: Odlišnosti v nastavení knihovny XStream pro formát XML a JSON

Ukázka jediné odlišnosti konfigurace pro změnu výsledného formátu je zobrazena v kódu 8. Hlavním krokem nastavení je určení třídy, která obsahuje anotace určující nastavení pro serializaci a deserializaci. .

```
@XStreamAlias("hierarchy-container")
public class DeviceHierarchy {

    @XStreamAlias("id")
    private long containerId;

    @XStreamAlias("name")
    private String containerName;

    @XStreamAlias("containers-set")
    private List<DeviceHierarchy> hierarchy = new ArrayList<>();
```

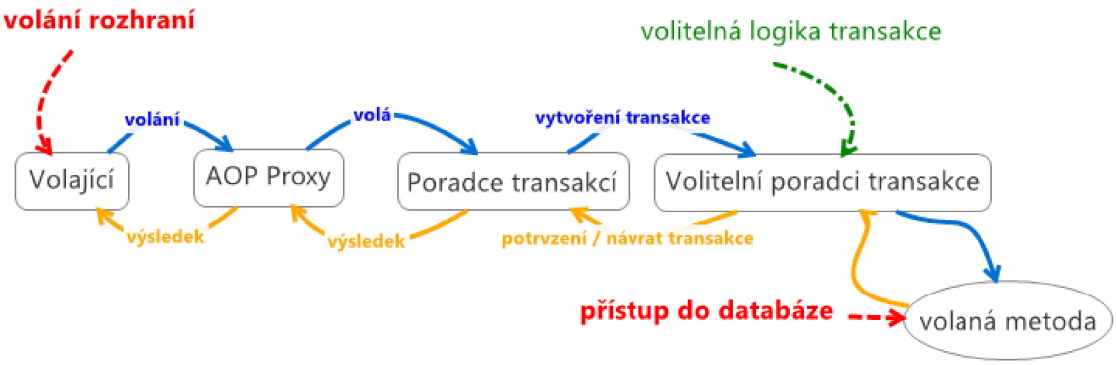
Kód 9: Třída s anotacemi pro serializaci pomocí XStream

Kód 9 popisuje implementace části třídy s nastavením pro knihovnu XStream. Anotace *@XStreamAlias* určuje název elementu ve výsledném formátu. Nevýhodou tohoto řešení je nutnost přesné znalosti výsledného formátu, jelikož XStream v případě výskytu neznámého elementu ukončí volání výjimkou.

4.1 Služby pro práci s databází

V aplikaci byla použita deklarativní správa transakcí rámce Spring. Veškeré dotazy na databázi jsou zapouzdřeny třídami obsahujícími logiku pro vytvoření transakce. Tyto třídy jsou označovány jako databázové služby (dále jen služby). Každá služba pracující s databází dědí v této aplikaci od třídy *DataService*, která poskytuje přístup k aktuální

session rámce Hibernate. *DataService* je součástí hierarchie i při dědění od generické třídy *AbstractDataService*, která navíc poskytuje základní metody pro práci s entitami. Volání metod jednotlivých služeb vytváří transakce, které jsou následně převedeny do databáze dle nastavení. Standardní nastavení rámce Spring vytváří novou *session* pomocí *SessionFactory* pro každé vlákno nebo dotaz. Takto vytvořená *session* trvá do doby ukončení poslední transakce. Ukončení transakce je provedeno jejím potvrzením nebo navrácením, případně pokud se během vykonávání vyskytne chyba, je provedena činnost dle konfigurace nastavené v rámci služby. Mezi standardní postup patří zapsání výjimky do záznamového souboru, navrácení transakce a následné ukončení vlákna zpracovávajícího požadavek.



Obrázek 6: Deklarativní správa transakcí rámce Spring [15]

Na obrázku 6 je znázorněno základní schéma volání služby v případě použití deklarativní správy transakcí. Místo volání cílové metody je nejdříve zavolána proxy cílové třídy. Dle nastavení se poté rekurzivně volají jednotlivé filtry. Základním filtrem je *TransactionAdvisor*, který na základě nastavení služby vytvoří, ukončí nebo pozastaví transakci. Po dokončení volání filtrů je v posledním kroku rekurze teprve volána metoda cílové služby. Po dokončení volané metody se vrací volání zpět v algoritmu.

4.1.1 Nastavení správy transakcí

Aby bylo možné využít výše uvedeného způsobu řízení transakcí, je třeba informovat aplikační kontext rámce Spring o přítomnosti rámce Hibernate pomocí konfiguračního souboru *persistence.xml*.

```
<!-- enable the configuration of transactional behavior based on annotations -->
<tx:annotation-driven transaction-manager="transactionManager"  />

<bean id="internalDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${dataSource.envis.internal.driver}" />
    <property name="url" value="${dataSource.envis.internal.url}"></property>
    <property name="username" value="${dataSource.envis.internal.username}" />
    <property name="password" value="${dataSource.envis.internal.password}" />
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="internalDataSource" />
    <property name="hibernateProperties" ref="dbProperties" />
    <property name="packagesToScan" value="net" />
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Kód 10: Nastavení deklarativní správy transakcí rámce Spring pomocí XML

V kódu 10 je hlavní část integrace rámce Hibernate do aplikačního kontextu rámce Spring. Nejprve je registrován zdroj dat. Každý zdroj dat musí obsahovat jméno, heslo, URL pro přístup do databáze a název ovladače. Mezi dalšími nepovinná nastavení patří maximální a minimální počet souběžně otevřených spojení. Během vývoje byl použit testovací zdroj dat, který neudržuje seznam spojení. Tento zdroj ignoruje nastavení počtu spojení, jelikož vytváří vždy nové připojení a jeho použití je určeno pouze pro testování. V reálném nasazení bude tento zdroj nahrazen za C3PO [15] s podporou pro správu udržovaných spojení. Na základě použitého zdroje dat je vytvořena *SessionFactory.* Dalším krokem je předání reference na *SessionFactory* správci transakcí, reprezentovanému rozhraním *TransactionManager. TransactionManager* určuje globální politiku pro práci s datovými transakcemi v celém modulu. V aplikaci byl použit správce dle specifikace JPA [18]. Správce JPA umožňuje základní přístup pomocí správce entit, který se plně řídí specifikací JPA a abstraktní přístup pomocí správy transakcí rámce *Hibernate*. Výhoda správce entit JPA spočívá v bezproblémové přenositelnosti, jelikož se plně řídí specifikacemi. V této aplikaci byl zvolen abstraktnější přístup pomocí správy transakcí rámce Hibernate z důvodu podpory pro dále zmiňované dotazovací jazyky. Posledním krokem byla registrace správce entit pro deklarativní správu transakcí přidáním XML elementu ze schématu TX. Takto nakonfigurovaný systém obaluje pomocí proxy

pro každou třídu a metodu anotovanou jako *@Transactional* vnitřní logiku a volání každé metody je provedeno během transakce. Podrobnější informace o možnostech a nastaveních jsou dostupné z manuálu [19].

```
@Transactional(isolation = Isolation.READ_COMMITTED)
@SuppressWarnings("unchecked")
@Service("deviceService")
public class DefaultDeviceService extends DataService implements DeviceService {
```

Kód 11: Použití anotací pro deklarativní správu transakcí

4.2 Použité způsoby pro dotazování do DB

Pro dotazování do databáze byly použity celkem tři způsoby. Mezi hlavní rozdíly patří různá abstrakce přístupu a s tím související kontrola efektivity vytvořených dotazů.

4.2.1 Jazyk SQL

Základní možností je použít jazyk SQL. Jazyk SQL pracuje na nejnižší úrovni z pohledu abstrakce, umožňuje plnou kontrolu nad tvarem výsledného dotazu a poskytuje zpravidla nejlepší výkonnost. Mezi hlavní nevýhody patří statická povaha dotazů a nemožnost kontroly dotazů již během kompilace. Přístup na této úrovni zcela ignoruje nastavení rámce Hibernate. Z důvodů rychlosti a co nejmenší provázanosti s databází byl tento způsob zvolen pro dotazování externí databáze. V nakonfigurovaném prostředí rámce Spring se dotaz provede na základě volání Spring JDBC šablony.

```
@Override
public MeasurementReference findMeasurement(long id) {
    return jdbcTemplate.getJdbcOperations()
            .queryForObject("select * from SmpMeasNameDB where Id = ?",
                    new MeasurementMapper(), id);
}
```

Kód 12: Dotazování databáze pomocí jazyka SQL

V kódu 12 je dotaz pomocí jazyka SQL. Samotná logika spojení a získání dat je skryta za rozhraním šablony rámce Spring. Výsledek je pomocí mapovací třídy převeden na objekt jazyka Java. Použitá metoda umožňuje základní kontrolu správnosti dat v externí databázi, jelikož dle její interní implementace určujeme očekávaný výsledek dotazu.

V případě volání metody *queryForObject( )* je očekáván maximálně jeden výsledek dotazu. Pokud je navrácen neunikátní výsledek nebo je zjištěna existence záznamu obsahujícího více sloupců se stejným názvem, dojde k vyvolání výjimky.

```
private List<ElmerConfig> listElmerConfig(List<Integer> list) {
    return jdbcTemplate.getJdbcOperations().query("select * from SmpConfigsDB " +
            "where Id in (:list)",
            new ElmerConfigMapper(),
            Collections.singletonMap("list", list));
}
```

Kód 13: Realizace ochrany proti SQL injektáži

Na úrovni jazyka SQL bylo nutné ošetřit parametry dotazu proti SQL injektáži. V kódu 12 je využita možnost předat parametry při vytváření SQL dotazu přímo jako parametr funkce. Tento způsob je vhodný pouze při malém množství parametrů a použití primitivních datových typů. V případě více parametrů nebo použití abstraktních datových typů je vhodné použít přístup implementovaný v kódu 13 nebo v kódu 2. Při volání metody je použita reference na mapu pojmenovaných SQL parametrů(kód 2), nebo je použita neměnná mapa jazyka Java (kód 13).

4.2.2 Jazyk HQL

Druhým způsobem je dotazovací jazyk HQL. Jedná se o objektovou nadstavbu jazyka SQL, řízenou konfigurací Hibernate, která navíc poskytuje podporu pro dědičnost, polymorfizmus a asociace [19]. Mezi hlavní výhody patří velmi snadná srozumitelnost a komplexní možnosti dotazů včetně podrobného nastavení způsobu jejich načtení. Hlavní nevýhodou je nutnost svázání podoby aplikační databáze s rámcem Hibernate a nemožnost kontroly dotazů při kompilaci. Popis možností jazyka HQL je mimo rámec této práce a je kompletně popsán na [20].

```
@Transactional(readOnly = true)
public Account findAccount(Person person, String password) {
    return (Account) getSession()
            .createQuery(
                    "from Account where owner = :person  AND passwordHash = :passwordHash")
            .setEntity("person", person)
            .setString("passwordHash", Hasher.hash(password))

            .uniqueResult();
}
```

Kód 14: Dotazování databáze pomocí jazyka HQL

Kód 14 znázorňuje jednoduchý dotaz pomocí jazyka HQL. Rozdíl oproti jazyku SQL je patrný v předaném parametru dotazu pomocí reference na již existující entitu, která je vyhodnocena při přípravě dotazu vyhledáním primárního klíče. Připravený dotaz je poté převeden do tvaru jazyka SQL a výsledek namapován do podoby požadovaného objektu. V této aplikaci byl jazyk HQL používán pro většinu dotazů do aplikační databáze.

4.2.3 Criteria API

Jedná se o dotazovací jazyk, který provádí dotaz vždy vůči určené entitě. Celý dotaz je reprezentován jako objekt obsahující sadu restrikcí, projekcí a nastavení, které jsou poté převedeny na jazyk SQL. Tento přístup již umožňuje kontrolu správnosti dotazu v době kompilace a snadné nastavení *cache* druhé úrovně. Jeho hlavní výhodou je možnost dotaz libovolně upravovat za běhu programu bez nutnosti nové kompilace. Hlavní nevýhodou je složitější vyjádření komplexních dotazů, například při sjednocování více tabulek, a zpravidla horší srozumitelnost pro programátory s menší znalostí jazyka SQL.

```
@Override
public Device findDevice(String deviceName) {
    return (Device) getSession().createCriteria(Device.class)
            .add(Restrictions.eq("name", deviceName)).setCacheable(true).uniqueResult();
}
```

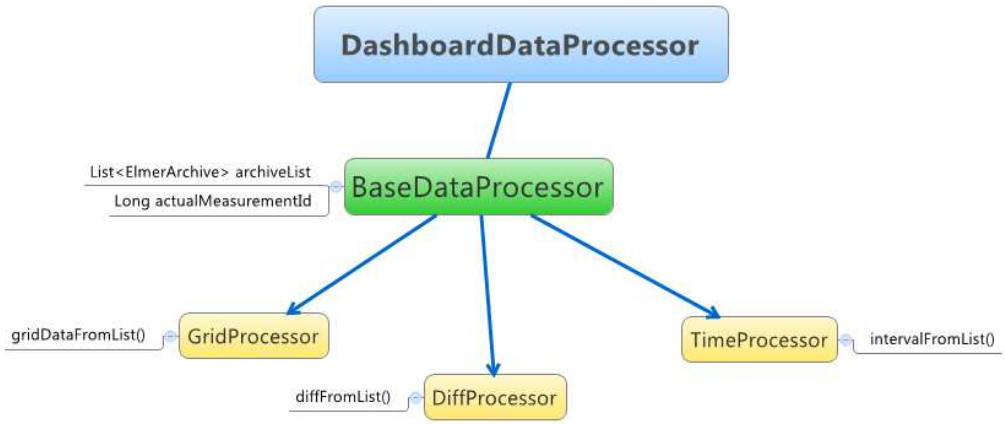
Kód 15: Dotazování databáze pomocí Criteria API

Dotaz pomocí Criteria API je znázorněn v kódu 15. Nejprve je vytvořena instance třídy Criteria na základě existující definice entity a poté jsou na ní aplikována určitá omezení. V dotazu je volána metoda *setCachable ( )* pro vytvoření vlastní *cache*

pro skupinu dotazů. Práce s *cache* v rámci Hibernate je řešena velice komplexně a modulárně. Jako *cache* lze zvolit vlastní implementaci nebo využít několik volně dostupných implementací. V této aplikaci byla použita *cache*, která je standardně součástí rámce Hibernate.

4.3 Logika zpracování dat z databází

S daty získanými z externí databáze je prováděno velké množství operací. Z tohoto důvodu byl vytvořen modulární návrh, který pomocí kompozice vytvoří požadovaný procesor dat. V aplikaci byly použity procesory pro zpracování času, rozdílu hodnot a dat tabulky. Každý procesor vychází z dat vytvořených pomocí instance třídy *BaseDataProcessor. BaseDataProcessor* obsahuje hodnoty získané z externí databáze, uložené jako seznam. Na obrázku 7 je ukázka použití kompozice pro získání instance třídy *DashboardDataProcessor,* kdy pomocí služby externí databáze je získána instance *BaseDataProcessor,* a ta je poté použita v konstruktoru třídy *DashboardDataProcessor,* který dále při operacích využívá logiku dalších procesorů k získání potřebných dat.



Obrázek 7: Hierarchie pro zpracování dat z externí databáze

5 Modul pro webovou prezentaci

Obrázek 8: Grafická podoba uvítací strany webové prezentace

Modul Appserver tvoří základní prvek v komunikaci mezi uživatelem a vnitřní logikou aplikace. Modul byl rozdělen do tří hlavních částí. První část tvoří logickou vrstvu rámce Struts2. Druhá část představuje prezentační vrstvu, tvořenou soubory jazyka JavaScript, kaskádových stylů a stránkami JSP. Jelikož tento modul tvoří část umístěnou na aplikační server, je zde přítomna i celková konfigurace systému včetně nastavení webového kontejneru.

5.1 Filtr pro přihlášení uživatele

Jelikož aplikace bude přístupná pouze registrovanému uživateli, bylo třeba vytvořit filtr pro přihlášení a validaci uživatele. Logika filtru je rozdělena na základě parametru určujícího, zdali data pocházejí z registračního formuláře, nebo se pouze jedná o obecný dotaz uživatele. Pokud data přicházejí z registračního formuláře, je dále testován parametr určující počet pokusů o přihlášení, aby bylo možné uživateli v případě několika

neúspěšných pokusů uživateli pomoci odkazem na odeslání nového hesla. Dále tento parametr slouží jako základní ochrana vůči mnoha souvislým pokusům o přihlášení.

V kódu 16 je implementace hlavní části logiky pro validaci uživatele. Nejprve je získán aktuální kontext akce, přes který je získán přístup k aktuální *session* uživatele. Pokud je v *session* přítomna reference na aktuálního uživatele pokračuje rekurzivní volání dalších filtrů. V opačném případě dojde k pokusu o přihlášení uživatele a na základě výsledku je vytvořena odpověď, určená řetězcem navráceným z filtru. V případě, že filtr navrátí textový řetězec, nebo skončí výjimkou, je odpověď vytvořena bez vyvolání dalších filtrů a výsledné akce.

```
@Override
public String intercept(ActionInvocation invocation) throws Exception {

    Map<String, Object> session = invocation.getInvocationContext().getSession();

    if ((Account) session.get(UserLoginStatics.USER_HANDLE) == null) {

        Map<String, Object> parameters = invocation.getInvocationContext().getParameters();
        String loginAttempt = (String) parameters.get(UserLoginStatics.LOGIN_ATTEMPT);

        if (StringUtils.isNotEmpty(loginAttempt)) {
            if (processLoginAttempt(parameters, session)) {
                return "login-success";
            } else {
                Object action = invocation.getAction();
                if (action instanceof ValidationAware) {
                    ((ValidationAware) action).addActionError("login.incorrect");
                }
            }
        }
        return "login";
    } else {
        return invocation.invoke();
    }
}
```

Kód 16: Implementace filtru pro přihlášení uživatele

K vyvolání filtru dojde na základě konfigurace rámce Struts2. Každý filtr je v rámci Struts2 identifikován svým unikátním názvem a na základě tohoto názvu lze filtr přiřadit do zásobníku filtru nebo přímo jednotlivým akcím.

```
<interceptor-stack name="defaultStack">
    <interceptor-ref name="exception">
        <param name="logEnabled">true</param>
        <param name="logLevel">DEBUG</param>
    </interceptor-ref>
    <interceptor-ref name="alias" />
    <interceptor-ref name="servletConfig" />
    <interceptor-ref name="i18n" />
    <interceptor-ref name="params">
        <param name="excludeParams">dojo\..*,^struts\..*</param>
    </interceptor-ref>
    <interceptor-ref name="login" />
```

Kód 17: Část zásobníku filtrů rámce Struts2

Umístění filtru pro přihlášení do zásobníku filtrů je znázorněno v kódu 17. Jelikož během jeho vyvolání nejsou žádně požadavky na předchozí filtry, není jeho pořadí v zásobníku pevně dáno. Zpravidla by ale měl být umístěn po filtru zpracovávajícím výjimky a před filtry, které pracují s logikou akce.

5.2 Vnitřní logika akce

Akce je základním prvkem rámce Struts2 a hlavní místo interakce modulu appserver s ostatními moduly. Díky použití filtrů je samotný kód přehledný a snadno testovatelný. Mezi základní akce této aplikace patří *DashboardAction*. Její hlavní úlohou je načtení dostupných skupin zařízení a nastavení aktuálního zařízení dle uživatelské konfigurace.

```
public class DashboardAction extends BaseAction implements Preparable, ModelDriven<DeviceRequestData> {

    private DashboardDataProcessor dashboardDataProcessor;
    private DeviceRequestData deviceRequestData;

    @Autowired(required=true)
    private ElmerServiceFacade elmerServiceFacade;
```

Kód 18: Hlavička definice akce rámce Struts2

DashboardAction dědí od třídy *BaseAction (kód 18)*, která slouží jako základní třída pro tvorbu dalších akcí. Hlavním úkolem *BaseAction* je poskytování přístupu k uživatelské *session* a k instanci přihlášeného uživatele. *BaseAction* dále dědí od třídy *ActionSupport*, která je již součástí rámce Struts2 a poskytuje základní rozhraní

pro validaci a lokalizaci dat, dále poskytuje statické hodnoty pro návratové hodnoty akce (přístupné na základě vnitřní implementace rozhraní *Action*). Použití kontextu rámce Spring pro injektování instance třídy implementující rozhraní *ElmerServiceFacade* je znázorněno v kódu 18.

```
public void prepare() throws Exception {
    if (super.hasSessionKey(DASHBOARD_DATA_PROCESSOR)) {
        this.dashboardDataProcessor = (DashboardDataProcessor) super.getSessionValue(DASHBOARD_DATA_PROCESSOR);
    }
    if (super.hasSessionKey(DEVICE_REQUEST_DATA)) {
        this.deviceRequestData = (DeviceRequestData) super.getSessionValue(DEVICE_REQUEST_DATA);
    }
}
```

Kód 19: Realizace úpravy dat před jejich samotným zpracováním

Jelikož *DashboardAction* implementuje rozhraní *Preparable*, následuje definice metody *prepare( )*. Metoda *prepare( )* je volána filtrem *PrepareInterceptor* před samotnou validací vstupních dat. V případě *DashboardAction* je jejím hlavním úkolem vyhledat objekty uložené v *session* na základě definovaných statických klíčů.

```
public void validate() {
    if (!super.isLoggedIn()) {
        this.addActionError(super.getText("user.nologin", "Please login first"));
    } else {
        if (deviceRequestData == null) {
            this.deviceRequestData = new DeviceRequestData(getActiveUserAccount().getOwner().getSettings());
        }
    }
}
```

Kód 20: Realizace validace dat před vykonáním logiky akce

Dalším krokem je volání metody v*alidate( )*. Metoda je definována na základě rozhraní *ValidationAware*, které implementuje třída *ActionSupport*. Nejprve je zkontrolováno přihlášení uživatele. I když je testování přihlášení realizováno již pomocí filtru, je vhodné provést kontrolu i v kódu samotné akce, jelikož každá akce může používat vlastní zásobník filtrů a mohlo by tedy dojít k odstranění *session* na základě nečinnosti uživatele. V případě nepřihlášeného uživatele je vytvořena chyba akce. Text chyby je určen na základě volání metody *getText( )*. První parametr je název klíče, podle kterého je v souborech s lokalizací vyhledán lokalizovaný text. V případě neexistující lokalizace nebo nedefinovaného klíče je použit standardní text, předaný jako druhý parametr metody.

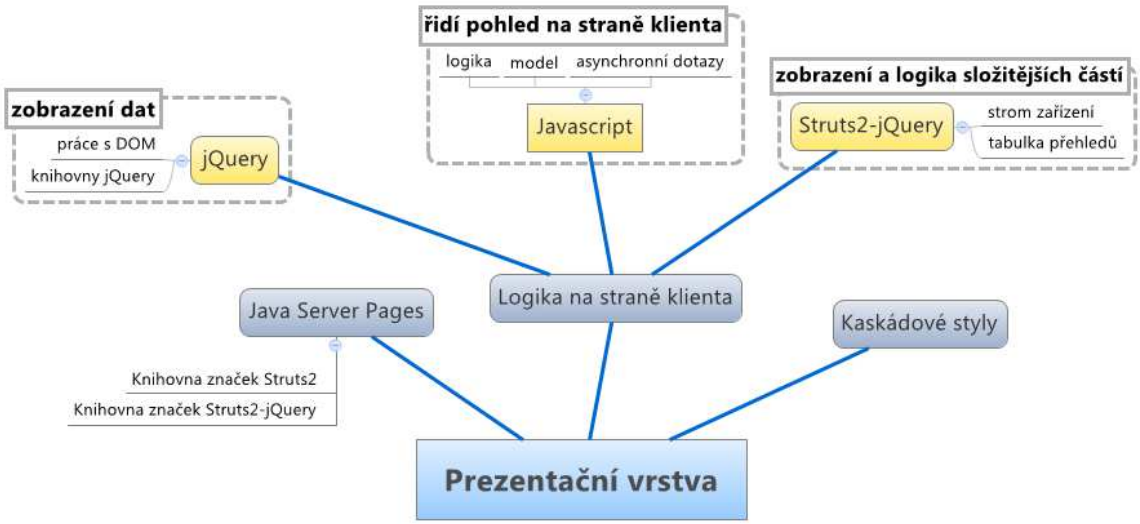
Pokud je uživatel přihlášen, dojde ke kontrole přítomnosti dat o zařízení pro vyhodnocení, v případě chybějících dat jsou použita data přidělená administrátorem. Tato situace řeší problematiku, kdy se uživatel přihlásí a nemá specifikované úvodní zařízení.

```
public String execute() throws Exception {

    if (dashboardDataProcessor == null) {
        dashboardDataProcessor = new DashboardDataProcessor(
                elmerServiceFacade.getDefaultDataProcessor(deviceRequestData.getMeasId()));
    }
    updateSession();

    return SUCCESS;
}
```

Kód 21: Realizace samotné výkonné logiky akce

Posledním krokem je volání metody *execute( )*. V tomto případě dojde k vytvoření procesoru dat elektroměru. Celá logika vytvoření procesoru je skryta za voláním metody rozhraní *ElmerServiceFacade* a je nezávislá na její vnitřní implementaci. Poté jsou získaná data uložena do *session* voláním metody *updateSession( )*. Výsledkem akce je textový řetězec určující výsledný pohled pro vytvoření odpovědi. V případě *DashboardDataProcesso*r jsou možné pouze dva pohledy. První je standardní pohled při hodnotě *SUCCESS* a druhý je chybový pohled. V aktuální fázi projektu je uživatel odeslán na stránku, umožňující zjistit příčinu vzniklé vyjímky.

5.3 Prezentační vrstva



Obrázek 9: Schéma návrhu a použitých technologií v prezentační vrstvě

Mezi základní komponenty prezentační vrstvy patří stránky Java Server Pages, soubory jazyka JavaScript a kaskádové styly.

5.3.1 Stránky JSP

Stránky JSP se z velké části skládají z HTML značek. Oproti klasickým HTML stránkám ale umožňují používat speciální knihovny značek. Mezi hojně využívané značky patří například *<jsp:include/>*, která umožňuje načtení určené stránky JSP do aktuální stránky. Hlavní předností technologie JSP je možnost použít předem definované knihovny jednotlivých rámců (Spring, Struts2). Knihovny mohou mít různou funkcionalitu od logické (podmínky, iterátor) až po dekorační. Jednotlivé značky mohou být nahrazeny i poměrně rozsáhlou šablonou. V aplikaci jsou použity knihovny rámce Struts2 a jQuery-Struts2. Knihovna Struts2 poskytuje hlavně logickou funkcionalitu a podporu pro přístup k zásobníku hodnot a kontextu akce. Knihovna jQuery-Struts2 [21] má především dekorativní a logický charakter pro vytváření složitějších grafických prvků, jako například stromu či záložek. Výhoda tohoto přístupu spočívá ve sjednoceném grafickém a logickém návrhu jednotlivých komponent.

```
<jsp:include page="./leftsidebar.jsp" />

<div id="content">
    <s:url id="chartDataUrl" action="createDeviceSimpleGraph" />
    <span id="actualMeasurementId" class="hidden">
    <s:property value="measId" /> </span>


    <span id="actualDeviceId" class="hidden">
    <s:property value="deviceId" /> </span> <span id="actualContainerId"
        class="hidden"><s:property value="containerId" /></span>
```

Kód 22: Použití značek z knihovny rámce Struts2 v kódu HTML

V kódu 22 je znázorněno použití značek Struts2. Značky rámce Struts2 lze identifikovat podle jmenného prostoru „*s*“. Například pomocí značky *<s:property/>* je přistupováno k hodnotám kontextu akce. K přístupu je použit jazyk OGNL [22]. V případě řetězce *measId* dojde k prohledání zásobníku hodnot na požadovaný klíč, chronologicky dle doby přidání objektu do zásobníku. Klíčem může být samotná hodnota, přístupová metoda nebo metoda obecně. Pomocí předpony „*#*“ lze navíc získat přístup i do ostatních částí kontextu akce (*session*, *parametry*).

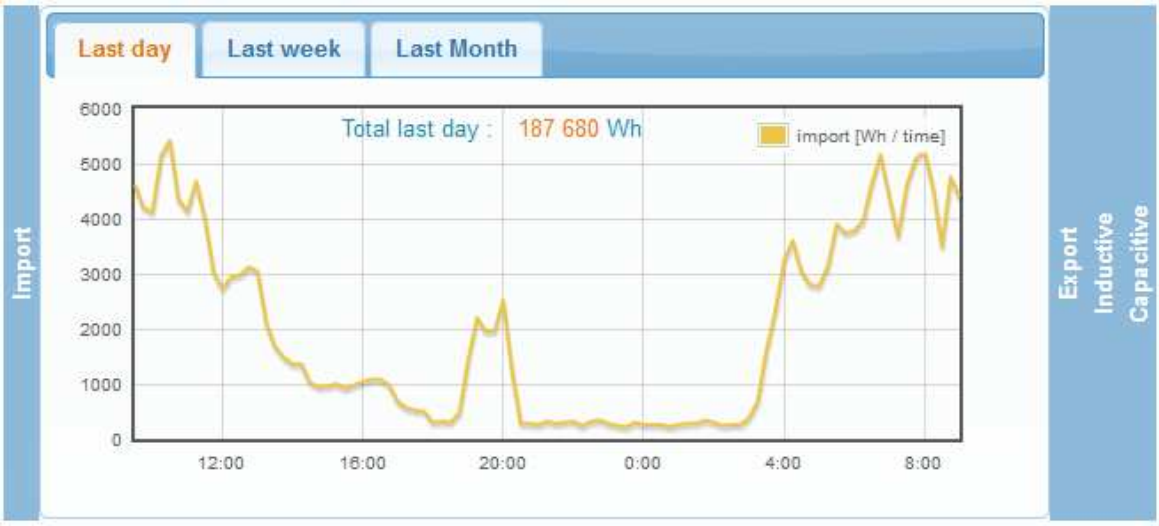
```
<?xml version="1.0" encoding="UTF-8"?>
<%@page contentType="text/html; charset=UTF-8" %>

<jsp:include page="../jsp/head.jsp"/>
<jsp:include page="../jsp/header.jsp"/>
<jsp:include page="../jsp/dashboard/dashboard.jsp"/>
<jsp:include  page="../jsp/footer.jsp"/>
```

Kód 23: Realizace složení výsledného pohledu pomocí technologie JSP

Pohledy v aplikaci jsou tvořeny z několika stránek JSP, které obsahují společnou část reprezentovanou stránkami *head.jsp*, *header.jsp* a *footer.jsp* (kód 23). Stránka *head.jsp* slouží k deklaraci hlavičky dokumentu a importu kaskádových stylů. Stránka *header.jsp* tvoří menu aplikace, které zůstává sjednocené v celé webové prezentaci pro usnadnění orientace uživatele. Poslední částí je stránka *footer.jsp*, která sjednocuje vzhled spodní části dokumentu a slouží jako místo pro vložení souborů jazyka JavaScript.

5.3.2 Soubory jazyka JavaScript

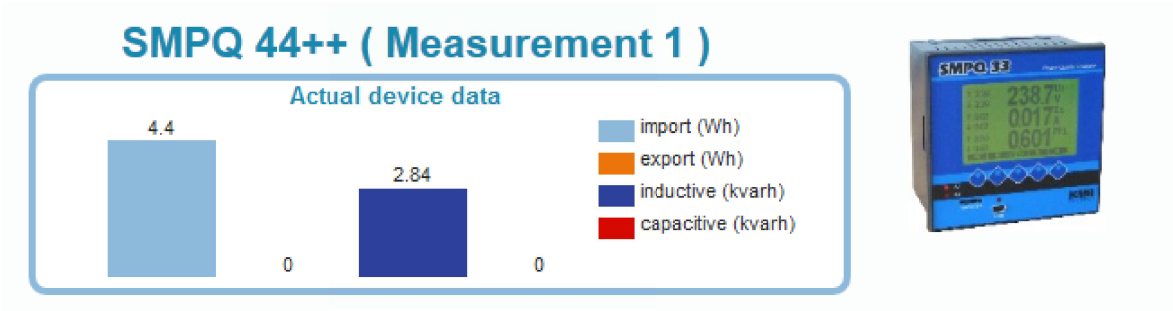
Jelikož aplikace zobrazuje data dynamicky na základě reakcí uživatele, je velká část logiky realizována pomocí jazyka JavaScript na straně klienta. Data jsou získávána pomocí asynchronních dotazů. Jelikož jazyk JavaScript nemá nativní podporu pro jmenné prostory a všechny proměnné jsou součástí globálního prostoru, byl pro oddělení jednotlivé funkcionality použit návrhový vzor Modul. Tento návrhový vzor skrývá jednotlivou funkcionalitu uzavřenou do modulů, které poskytují pouze přístupové atributy. Celý princip návrhového vzoru Modul spočívá ve vytvoření globální proměnné, ve které je uchováván výsledek volání anonymní funkce zvané c*losure* [23]. *Closure* si uchovává reference na všechny lokální proměnné vytvořené během volání funkce. Výsledkem je poté objekt globálního prostoru, s veřejnými atributy. Výhodou tohoto přístupu je odstínění vytvořených proměnných z *closure* od globálního prostoru.



Obrázek 10: Grafický přehled sledovaných hodnot pro zvolené zařízení

Pro práci s DOM je použit rámec jQuery [24]. V aplikaci je využíván především pro vyhledávání elementů a jako základ pro integraci se zásuvnými moduly pro tvorbu grafického prostředí.

Na obrázku 10 je znázorněna realizace komponenty pro přehled sledovaných hodnot vybraného zařízení. Data pro vykreslení grafu jsou získána na základě asynchronního volání logiky jazyka JavaScript ve formátu JSON. Záložky jsou realizovány pomocí knihovny jQuery a slouží pro přepínání mezi sledovanými intervaly.



Obrázek 11: Aktuální hodnoty odečtu zvoleného zařízení

5.4 Konfigurace systému

Pří kompilaci jsou nejprve sestaveny moduly Core a Core_External a poté jsou poskytnuty jako knihovny při kompilaci modulu Appserver. Výsledkem sestavení je WAR, který je poté umístěn do webového kontejneru. Aby bylo možné s aplikací komunikovat přes webový kontejner, je nutné nastavit popis pro umístění, reprezentovaný souborem web.xml.

5.5 Popis pro umístění aplikace na server

```
<web-app id="starter" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Envis</display-name>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:applicationContext*.xml</param-value>
    </context-param>

    <context-param>
        <param-name>spring.profiles.active</param-name>
        <param-value>DEVEL,DB</param-value>
    </context-param>
```

Kód 24: Nastavení umístění aplikačního kontextu v souboru web.xml

Soubor *web.xml* se skládá ze sady deklarací (kód 24, 25, 26). První část tvoří deklarace elementu *web-app* s verzí servletů a jmenného prostoru jazyka XML. Dalšími elementy jsou název webového archivu a parametry kontextu. Parametry nastavují globálně dostupné informace pro celou aplikaci a mezi hlavní části patří deklarace umístění aplikačního kontextu rámce Spring.

```
<filter>
    <filter-name>action2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>

<filter-mapping>
    <filter-name>action2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Kód 25: Integrace rámce Struts2 do webového kontejneru

Následuje nastavení filtrů rámce Struts2. Nejprve je vytvořen filtr rámce Struts2 a poté je namapován na tvar výsledné adresy, pro kterou má být použit (kód 25).

```
<!-- Listeners -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- Welcome file lists -->
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Kód 26: Integrace rámce Spring do webového kontejneru s přehledem uvítacích stránek

Poslední částí je nastavení posluchače pro načtení XML konfigurace do aplikačního kontextu rámce Spring a volitelný seznam stránek sloužících jako vstupní bod do aplikace (kód 26). Tímto je nastavení popisu pro umístění na server dokončeno a aplikaci je možné po sestavení umístit na server.

5.5.1 Realizace aplikačního kontextu rámce Spring

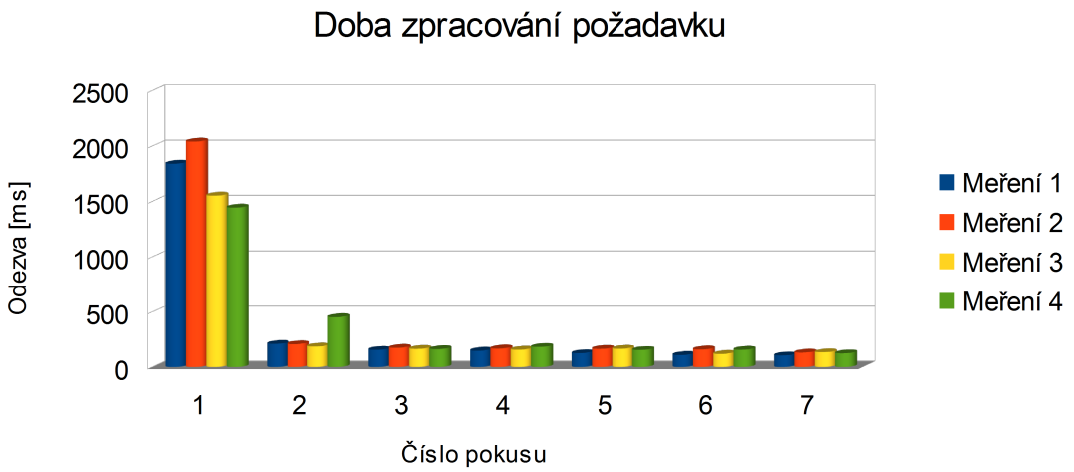
Při nastavení popisu pro umístění bylo odkazováno na aplikační kontext rámce Spring. Jedná se o soubor obsahující veškeré nastavení a deklarace rámce Spring. Jelikož se skládá z velkého množství deklarací, je v aplikaci rozdělen na několik souborů, pojmenovaných podle funkcionality, se kterou se pojí. Toto rozdělení umožňuje snazší a přehlednější správu jednotlivé funkcionality. V případě změny funkcionality stačí pouze vyměnit zvolený soubor.

```
<context:component-scan base-package="net.envis" />
<aop:aspectj-autoproxy />
<context:spring-configured />
<context:property-placeholder
    location="classpath:db-devel.properties,classpath:app-devel.properties" />

<util:properties id="dbProperties" location="classpath:db-devel.properties" />
<util:properties id="appProperties" location="classpath:app-devel.properties" />
```

Kód 27: Konfigurace aplikačního kontextu pomocí XML

Hlavní část nastavení je zobrazena v kódu 27. Vysvětlení jednotlivých bodů konfigurace je mimo rámec této práce a je podrobně popsáno v [19].Mezi hlavní body patří vyhledání všech anotovaných definic tříd a jejich přiřazení do kontextu. Dále zvolené nastavení umožňuje použití anotací *@Autowired* i mimo třídy spravované rámcem Spring pomocí aspektově orientovaného programování. Dále jsou v nastavení vyhledány konfigurační soubory, které jsou načteny do kontextu a přístupné na základě jejich názvu. Poté již následuje pouze import jednotlivých dílčích souborů s nastaveními.

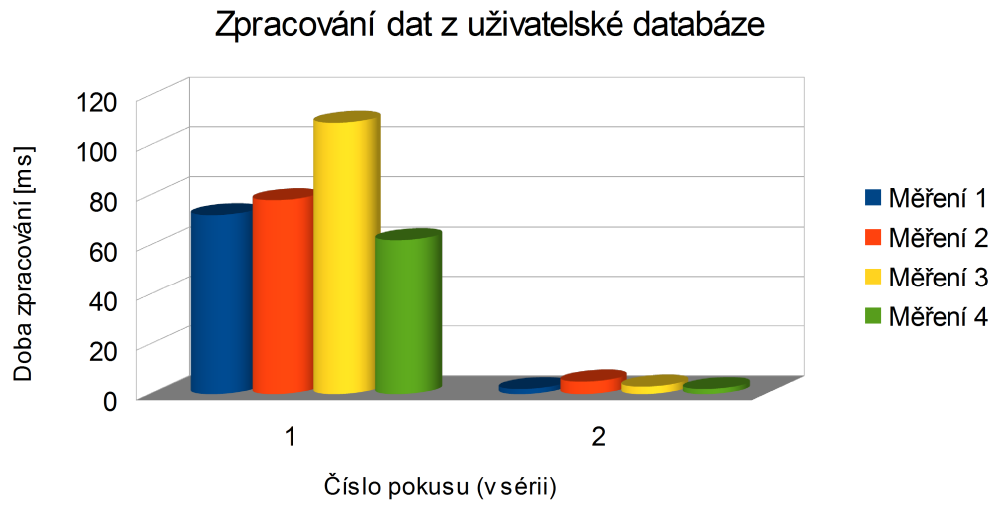
5.6 Doba potřebná pro zpracování požadavku



Graf 1: Doba zpracování uživatelského požadavku

V grafu je zobrazena odezva aplikace při zobrazení uvítací strany po přihlášení uživatele. V tomto stavu aplikace získává a zpracovává data za poslední měřený týden. Z grafu je možné pozorovat výrazné snížení doby odezvy při následných dotazech, jelikož

jsou data ukládána do *cache* na několika úrovních a zpracovaná data poté uložena do uživatelské *session*. Testování probíhalo lokálně, kdy na počítači byl nainstalována samotná aplikace, aplikační server Apache Tomcat 7 a databázový server MySQL 5.2.



Graf 2: Doba zpracování dat z uživatelské databáze

Samotné měření má spíše informativní charakter o efektivitě návrhu a z důvodů nedostatečné infrastruktury nebylo provedeno na samostatném aplikačním serveru. Pro poskytnutí odpovídajících výsledků by bylo nutné mít k dispozici alespoň dva servery s odlišnou konfigurací. Ke zkreslení výsledků při testech navíc přispívalo samotné umístění veškeré potřebné infrastruktury na lokálním počítači, na kterém běžely během testu další aplikace, včetně vývojového prostředí. Umístěním veškeré infrastruktury na jediném počítači způsobovalo odchylky v důsledku náhodném zaneprázdnění systému při zpracování požadavků ostatních aplikací, především při přístupu na pevný disk. Pro upřesnění výsledků by bylo vhodné simulovat přístup několika uživatelů zároveň. Větší počet požadavků uživatelů má negativní vliv na rychlost z důvodu obsluhy paralelních požadavků, jak na úrovni vnitřní logiky, tak na úrovni přístupu do databází. Na druhé straně ale umožňuje také zrychlení zpracování některých požadavků, kdy jsou do výsledků promítnuty předzpracované výsledky uložené v *cache*, jak na úrovni rámce Hibernate, tak na úrovni rámce Spring a uživatelské *session*.

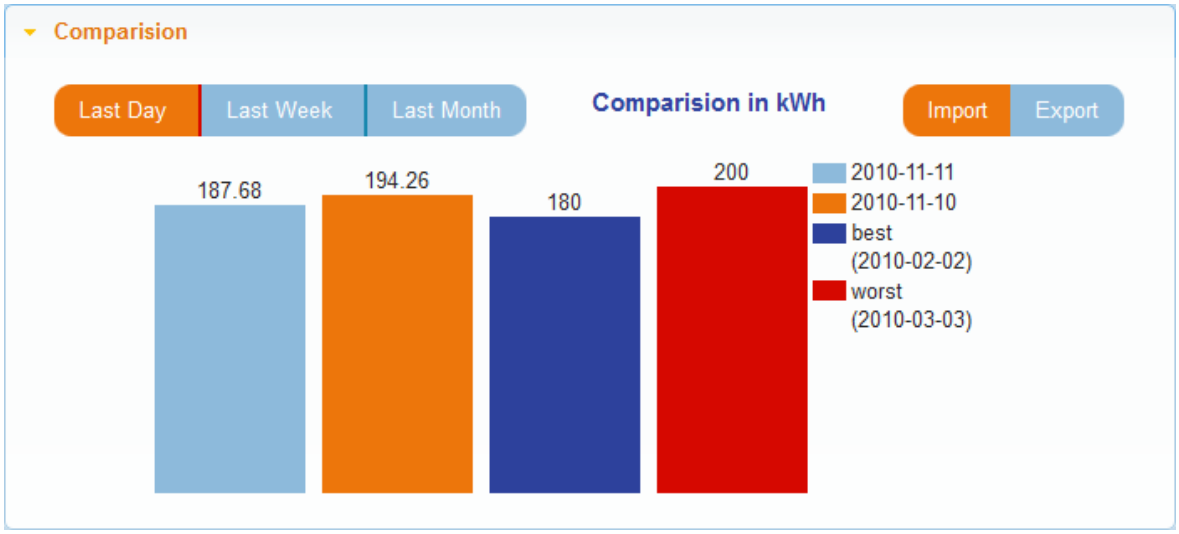
6 Vyhodnocení výsledné funkcionality aplikace

Aplikace se nyní nachází ve verzi, která byla specifikována během analýzy. Jelikož byl vývoj aplikace v první části proveden pouze na omezených datech obsahujících pouze dvě zařízení s jedním měřením, jsou některé části uměle vytvořené pro potřeby prezentace a testování. Mezi uměle vytvořené části patří hlavně strom zařízení a výčet dostupných zařízení, který nyní obsahuje velký počet měření, která ale mají stejnou hodnotu identifikátoru. Celkové hodnocení funkčnosti a vzniklých problémů je logicky rozděleno do tří částí, které v některých případech neodpovídají přímo rozdělení na jednotlivé moduly, ale spíše odpovídají rozdělení na jednotlivé domény funkčnosti.

6.1 Uživatelské rozhraní

Obrázek 12: Komponenty uvítací stránky webové prezentace

Uživatelské rozhraní (obrázek 12) umožňuje výběr požadovaného měření zařízení ze stromové struktury (část označená jako 3), uložené v aplikační databázi. Pro vybrané zařízení se zobrazí aktuální odečty a grafické zobrazení zařízení (část 2). Dále je možné sledovat grafy jednotlivých sledovaných veličin za zvolená období (část 1), vytvořených na základě dat získaných asynchronně pomocí technologie AJAX



Obrázek 13: Porovnání naměřených hodnot pro zvolené zařízení

Pro zvolené zařízení je poté možno provést porovnání vybraných období, data jsou získávána asynchronně a při akci uživatele dojde k okamžitému překreslení grafu bez nutnosti načítat znovu celý obsah webové stránky.

Elmer Data

Time	Import [kW]	Export [kW]	Capacitive [kvarh]	Inductive [kvarh]
12.11.2010 06:30	3.9	0	0	4.08
12.11.2010 06:15	3.14	0	0	2.82
12.11.2010 06:00	2.78	0	0	2.04
12.11.2010 05:45	2.8	0	0	1.7
12.11.2010 05:30	3.04	0	0	1.98
12.11.2010 05:15	3.6	0	0	2.04
12.11.2010 05:00	3.24	0	0	1.84
12.11.2010 04:45	2.32	0	0	1.06
12.11.2010 04:30	1.62	0	0.04	0.58
12.11.2010 04:15	0.68	0	0.02	0.16
12.11.2010 04:00	0.38	0	0.02	0
12.11.2010 03:45	0.26	0	0.08	0
12.11.2010 03:30	0.26	0	0.08	0
12.11.2010 03:15	0.24	0	0.06	0
12.11.2010 03:00	0.3	0	0.04	0.02

Page 1 of 13 15

Obrázek 14: Tabulkový výpis jednotlivých odečtů elektroměru

Uživatel má možnost si jednotlivé odečty prohlédnout v tabulkovém výpisu a v případě potřeby je možné stáhnout zprávu za zvolené období ve formátu PDF. Realizace tabulkového výpisu i samotného vytvoření dokumentu je realizováno pomocí asynchronních volání a uživatel není nucen čekat na samotné zpracování požadavku.

Obrázek 15: Komponenta pro uložení zprávy o zařízení ve formátu PDF

Funkcionalita aplikace je závislá na povoleném jazyce JavaScript ve webovém prohlížeči. Z hlediska přístupnosti aplikace pro různé platformy a nastavení prohlížečů by bylo vhodnější mít verzi pro uživatele bez možnosti povolit JavaScript ve svém prohlížeči. Vývoj paralelních verzí uživatelského rozhraní je ale časově náročnější a vývoj by musel být na úkor implementace další funkcionality. Jelikož je aplikace zaměřena na velmi omezenou skupinu konkrétních uživatelů, nejedná se o značnou nevýhodu, protože bude vyvážena mnohem interaktivnějším uživatelským prostředím. Diskutabilní je i samotné procento uživatelů bez možnosti povolit JavaScript ve svém prohlížeči. Tato možnost je aktuálně dostupná ve všech rozšířených prohlížečích a i ve většině nových mobilních zařízení, včetně tabletů a čteček elektronických knih.

Jelikož byla práce zaměřena více na vnitřní logiku, než na samotnou grafickou podobu uživatelského rozhraní, nebylo provedeno kompletní testování pro různé typy prohlížečů a typů platforem. Během vývoje byla aplikace otestována na prohlížečích Mozilla Firefox ve verzi 11.0, Google Chrome verze 17 a Internet Explorer 9 a nebyly zjištěny žádné změny v grafické prezentaci, které by neumožňovaly aplikaci používat. I když je aplikace primárně určena pro zobrazení na monitorech s vyšším rozlišením, je její budoucí přizpůsobení návrhu mobilním prohlížečům logickým krokem.

V další fázi vývoje bude do aplikace implementována logika pro ukládání výsledků asynchronních volání do lokální *cache*. Tímto bude efektivně snížen počet požadavků například při procházení tabulky s detailním výpisem měření nebo během porovnávání hodnot jednotlivých měření.

Vytváření uživatelských pohledů je nyní realizováno pomocí stránek JSP, ale v budoucnu lze jednoduše použít šablony rámce Freemarker nebo Velocity. Celý návrh zpracování požadavku je realizován velmi pružně a je možné jej jednoduše konfigurovat. Například úpravu zpracování dat ze standardního dotazu ve tvaru html/text do tvaru JSON (pro asynchronní volání) lze provést pomocí dvou jednoduchých kroků, bez zásahu do logiky samotné akce. Aplikace má integrovanou podporu pro internacionalizaci textu, která se ale aktuálně nachází pouze ve stavu funkčního návrhu, s absencí textů.

6.2 Vnitřní logika

Jednotlivé části aplikace mají stanoveny rozhraní pro komunikaci. Samotná implementace je realizována s dodržováním základních pravidel pro vývoj a je použito velké množství návrhových vzorů: Factory, Singleton, Facade, Command a Proxy. V aplikaci jsou použity možnosti aspektově orientovaného programování a v budoucnosti je plánováno jeho další rozšíření i mimo možnosti rámce Spring. Některé části systému jsou otestovány pomocí jednotkových testů a během dalšího vývoje budou doplněny jednotkové testy pro ověření funkčnosti většiny komponent.

Aplikace využívá aplikačního kontextu rámce Spring a tím minimalizuje závislosti mezi jednotlivými částmi systému. Části systému jsou rozděleny do funkčních bloků pomocí konfiguračních souborů, které je možno jednoduše změnit bez porušení funkcionality celé aplikace. Konfigurace systému je velice modulární a umožňuje bezproblémové přidání dalších částí. Bez zásadních změn v návrhu je možné ji kdykoliv rozšířit o vyšší úroveň zabezpečení na úrovní uživatelských rolí pomocí modulu Spring Security nebo přidáním podpory pro odesílání zpráv od emailů až po zprávy mezi komponentami například pomocí Java Message Service [25].

Na úrovni přístupu dat byl vytvořen návrh na základě rámců Hibernate a Spring. Datová vrstva byla otestována na databázích MySQL, MSSQL a PostgreSQL a lze ji použít bez nutnosti zásadně měnit nastavení aplikace, zpravidla stačí změnit jen ovladač databáze a přihlašovací údaje. Při přístupu k databázi je použito několik úrovní abstrakce, kdy přístup k externí databázi je prováděn pomocí šablony rámce Spring a SQL dotazů. Přístup do aplikační databáze používá objektově-relační mapování a přístup pomocí jazyků HQL a Criteria API rámce Hibernate. Integrace rámce Spring a Hibernate je nastavena s co nejmenšími závislostmi.

Data získaná z externí databáze jsou zpracovávána pomocí kompozice procesorů dat, které poskytují potřebnou funkcionalitu pro zobrazení dat uživateli. Jelikož získání dat z externí databáze je časově náročné, jsou vytvořená data uchovávána v uživatelské *session*, což s sebou přináší větší nároky na velikost paměti serveru, kdy procesor dat může mít až několik MB dat. Tento problém bude v budoucnu vyřešen pomocí vlastní implementace *cache* na úrovni volání metod modulu Core.

6.3 Správa a sestavení aplikace

Aplikace je uložena v privátním úložišti a je sestavena pomocí správce sestavení Apache Maven. Díky tomuto přístupu je možno vyvíjet aplikaci v různých podporovaných vývojových prostředích bez nutnosti měnit nastavení projektu. V hlavním abstraktním modulu je umístěn soubor *pom.xml*, který deklaruje globální závislosti na ostatní knihovny a moduly. V každém modulu aplikace je nastaveno chování při sestavení a testování aplikace. Nastavení systému Maven provede při každém sestavení všechny přítomné jednotkové testy a v případě jejich selhání nedojde k úspěšnému sestavení. Dále je zde umístěn plugin pro jednoduché vzdálené umístění na server a také pro vygenerování dokumentace k aplikaci.

Závěr

Cílem bakalářské práce bylo navrhnout modulární webovou aplikaci pro zpracování dat o využití elektrické energie. Vývoj probíhal na základě spolupráce s firmou KMB systems. V počáteční fázi vývoje byla provedena analýza požadavků potenciálních zákazníků, zhodnocena funkcionalita aktuálního softwarového řešení firmy a dostupných konkurenčních řešení. Na základě získaných údajů byl proveden návrh základní funkcionality a vytvořen základní plán vývoje. Během návrhu byla z důvodu co nejlepší přenositelnosti zvolena platforma Java EE. V aplikaci jsou začleněny aktuální technologie, používané v moderních Java EE webových aplikacích. Jako základní technologie pro tvorbu aplikace byly zvoleny aplikační rámce Spring, Struts2 a Hibernate, které dnes patří k velmi populárním a žádaným v komerční sféře.

Celý systém byl rozdělen do tří modulů, které mezi sebou komunikují na základě předem definovaných rozhraní. Logika uvnitř modulů je díky rámci Spring bez vzájemných závislostí mezi komponentami, rozdělených do samostatně konfigurovatelných funkčních bloků.

Aplikace pro svoji činnost využívá interní databázi, s podporou pro většinu rozšířených poskytovatelů databází. Přístup do interní databáze je řešen abstraktně pomocí objektově relačního mapovaní, které umožňuje pružně reagovat na změny v návrhu. Data z jednotlivých elektroměrů jsou získávána z uživatelské databáze, na kterou byly kladeny co největší požadavky z hlediska nezávislosti návrhu na typu dat. Změnu poskytovatele nebo tvaru tabulek databáze lze provést pouze změnou nastavení a mapovacích tříd uvnitř modulu Core_External.

Pro webovou prezentaci byla použita technologie JSP. Samotný pohled na webovou prezentaci byl vytvořen na základě nezávislých komponent. Logika na straně klienta byla realizována pomocí jazyka JavaScript. Klient z velké části komunikuje se serverem na základě technologie AJAX, která umožňuje reagovat na akce uživatele bez nutnosti načítat znovu celý obsah webové stránky a umožňuje efektivnější a uživatelsky přívětivější realizaci webové prezentace.

Aplikace se nyní nachází ve stavu funkčního prototypu, obsahující funkcionalitu navrženou během analýzy, tvořící kvalitní základ pro její budoucí rozšiřování. Aplikace umožňuje vyhodnocovat a porovnávat data ve zvolených intervalech a vytvořit zprávu

o vybraném zařízení. Pomocí stromového zobrazení zařízení byla vyřešena problematika zobrazení různého množství dostupných zařízení. Realizace komponent byla přizpůsobena dostupné sadě dat v testovací databázi a v budoucnu bude třeba logiku aplikace otestovat na větším množství dat. Mezi hlavní nedostatky aplikace patří chybějící administrační rozhraní, které bude obsahovat poměrně komplexní logiku a proto bylo rozhodnuto ji implementovat až v dalším kroku vývoje. Mezi další části patří realizace vzdáleného nahrávání dat do uživatelské databáze a možnost vytvoření detailnějších zpráv o zvoleném zařízení. Pro výše uvedenou logiku by bylo vhodné použít rámec DWR, který umožňuje mimo jiné asynchronní nahrávání a stahování souborů.

Systém správy sestavení aplikace z vytvořených modulů byl realizován pomocí konfigurace systému Maven. Vytvořené nastavení umožňuje jednoduché sestavení aplikace s automatickým získáním všech potřebných knihoven a zásuvných modulů. Nastavení systému Maven dále umožňuje snadné vytvoření dokumentace, hromadné spuštění jednotkových testů a vzdálené umístění aplikace na server. V budoucnu je v plánu do systému Maven začlenit modul pro kompresi souborů jazyka JavaScript a kaskádových stylů při kompilaci. Důvodem pro kompresi souborů je snížení množství dat přenášených na klienta při načítání webové prezentace.

Seznam použité literatury

[1] *KMB systems* [online]. 2011 [cit. 2012-05-06]. Dostupné z:http://www.kmb.cz/index.php/cs/

[2] DEVELOPER EXPRESS INC. *DXperience* [online]. [cit. 2012-05-13]. Dostupné z: http://www.devexpress.com/Subscriptions/DXperience/

[3] Energy management portal. NORTHERN DESIGN (ELECTRONICS) LTD. *ND Meter* [online]. [cit. 2012-05-06]. Dostupné z: https://www.energiaccount.com/TRIAL/login.php

[4] *SMeasure*: *Online software making building energy management easy for business* [online]. 2012 [cit. 2012-05-12]. Dostupné z: http://smeasure.com/

[5] *OPOWER*: *Online Energy Management Tools* [online]. 2012 [cit. 2012-05-12]. Dostupné z: http://opower.com/what-is-opower/analytics_portal

[6] ORACLE. *Java EE* [online]. [cit. 2012-05-12]. Dostupné z: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[7] *.NET Framework Developer Center* [online]. 2012 [cit. 2012-05-14]. Dostupné z: http://msdn.microsoft.com/en-us/netframework/aa496123

[8] Java EE Productivity Report 2011. ZEROTURNAROUND. *ZeroTurnaround* [online]. 2011 [cit. 2012-05-13]. Dostupné z: http://zeroturnaround.com/java-ee-productivity-report-2011/

[9] ORACLE. *Java Community Process*: *JSR 315: JavaTM Servlet 3.0 Specification* [online]. 2012 [cit. 2012-05-14]. Dostupné z: http://jcp.org/en/jsr/detail?id=315

[10] THE APACHE SOFTWARE FOUNDATION. *Apache Tomcat* [online]. 2012 [cit. 2012-05-12]. Dostupné z: http://tomcat.apache.org/

[11] *SpringSource.org* [online]. [cit. 2012-05-06]. Dostupné z: http://www.springsource.org/

[12] *Struts 2* [online]. [cit. 2012-05-06]. Dostupné z: http://struts.apache.org/2.2.1/index.html

[13] *Apache Maven* [online]. [cit. 2012-05-06]. Dostupné z: http://maven.apache.org/

[14] Relational Persistence for Java and .NET. THE APACHE SOFTWARE FOUNDATION. *Hibernate - JBoss Community* [online]. [cit. 2012-05-06]. Dostupné z: http://www.hibernate.org/

[15] *Spring in Action*. THIRD EDITION. Shelter Island, NY: Manning Publications Co., 2011. ISBN 9781935182351.

[16] *XStream* [online]. 2011 [cit. 2012-05-08]. Dostupné z: http://xstream.codehaus.org/

[17] *C3p0:JDBC DataSources/Resource Pools* [online]. 2011 [cit. 2012-05-08]. Dostupné z: http://sourceforge.net/projects/c3p0/

[18] ORACLE. *Java Community Process*: *JSR-000220 Enterprise JavaBeans 3.0* [online]. 2011 [cit. 2012-05-14]. Dostupné z: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[19] *Spring Framework*: *Reference Documentation* [online]. [cit. 2012-05-06]. Dostupné z: http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/

[20] *Java Persistence with Hibernate*. REVISED EDITION OF HIBERNATE IN ACTION. New York, NY: Manning Publications Co., 2007. ISBN 1-932394-88-5.

[21] *Struts2-jquery* [online]. [cit. 2012-05-06]. Dostupné z: http://code.google.com/p/struts2-jquery/

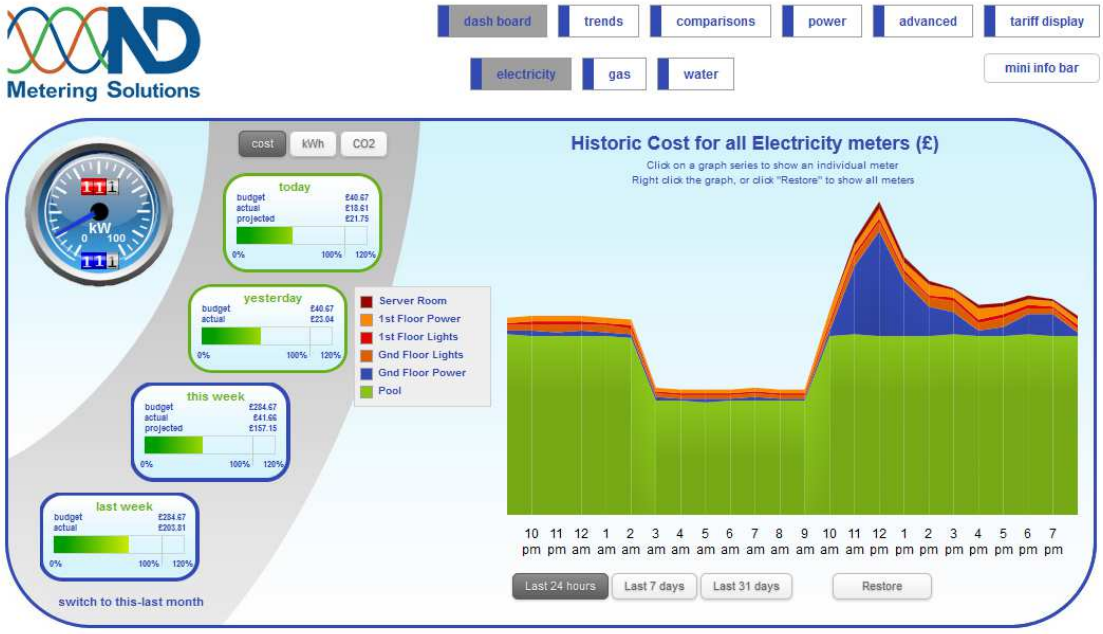
[22] *Struts2 in Action*. Manning Publications Co., 2008. ISBN 1-933988-07-X.

[23] *JavaScript*: *The Definitive Guide*. Sixth Edition. Sebastopol: O'Reilly Media, Inc., 2011. ISBN 978-0-596-80552-4.

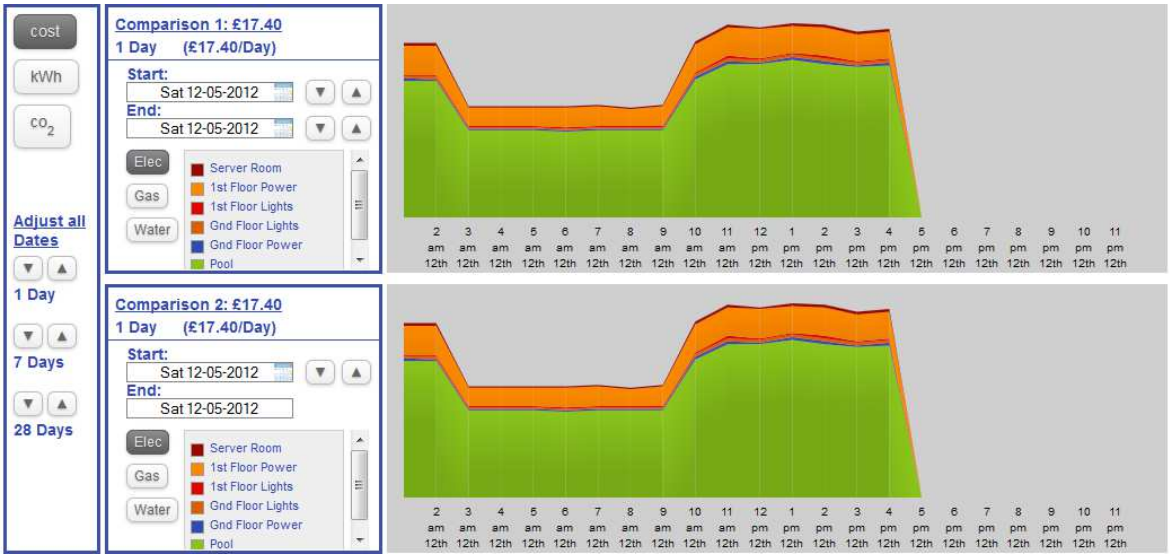
[24] *JQuery* [online]. 2012 [cit. 2012-05-06]. Dostupné z: http://jquery.com/

[25] JSR-000914 JavaTM Message Service (JMS) API. ORACLE. *Java Community Process* [online]. 2011 [cit. 2012-05-14]. Dostupné z: http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html

Příloha A : Řešení firmy ND Meter



Obrázek 16: Vzhled webové aplikace firmy ND Meter



Obrázek 17: Porovnání jednotlivých měření ND Meter

Příloha B : Popis pro zprovoznění aplikace

1. Nainstalovat webový server s podporou Java Servlet API v3.0 (Apache Tomcat 7)
2. Nainstalovat databázový server MSSQL
3. Vytvořit databázové schéma Envis_External a Envis_Internal
4. Vytvořit uživatele *„envis“* (heslo : *„envis“*) a dát mu plná práva k vytvořeným schématům
5. Nastavit MSSQL tak, aby bylo možné komunikovat přes následují spojení:
   1. `jdbc:jtds:sqlserver://localhost:1433/Envis_External`
   2. `jdbc:jtds:sqlserver://localhost:1433/Envis_Internal`
6. Nainstalovat aplikaci ENVIS a exportovat testovací data do Envis_External
7. Umístit envis.war (přiložen na CD) do kontejneru aplikačního serveru
8. Do aplikace se lze přihlásit na adrese : http://localhost:8080/appserver/index.action
9. Automaticky je vytvořen testovací uživatel s následujícími údaji : jméno : „envis“, heslo : „debug“